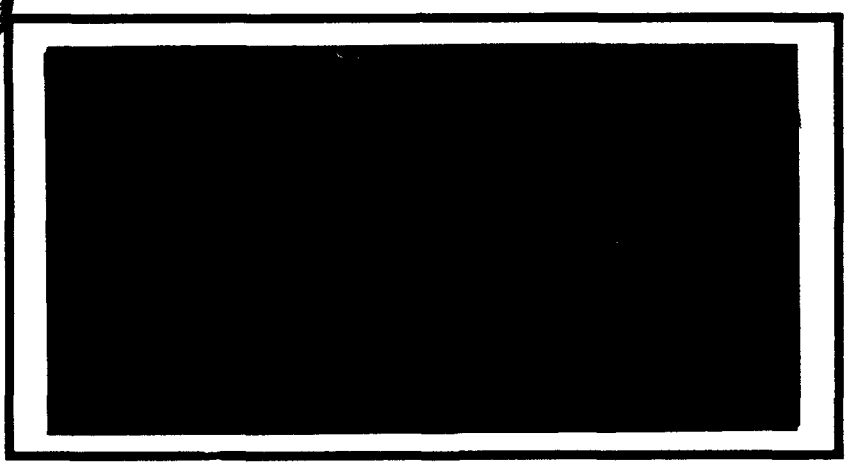
(1)



DEPARTMENT OF THE AIR FORCE

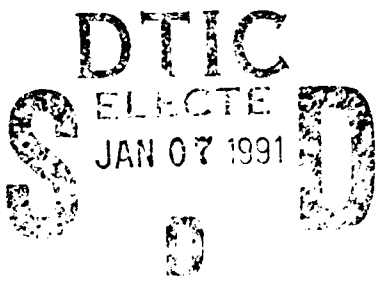
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

(1)

AFIT/GE/ENG/90D-41



Specification and Equivalence Verification of Sequential Circuits via VHDL

THESIS

Richard L. Miller
Captain, USAF

AFIT/GE/ENG/90D-41

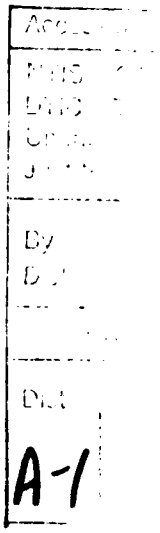
Specification and Equivalence Verification of Sequential Circuits via VHDL

Thesis

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Richard L. Miller, BSEE
Captain, USAF

November, 1990



J

Preface

This research presents a merger of the specification and design capabilities of the VHDL with a known verification method in order to solve the design and verification problem of sequential circuits. The fruits of this research are a behavioral VHDL model for sequential circuit specification, a structural VHDL model for sequential circuit design, and a method for comparing two circuits described using these VHDL models in order to demonstrate circuit equivalence.

In performing this research I received immeasurable assistance from many people. At this time, I would like to thank my advisor, Capt Bruce George, for his exceptional counsel and leadership in directing the nature of my research. Additionally, thanks go to Capt Mark Mehalic and Major Kim Kanzaki; their guidance and comments surely improved the quality of the finished product. I would also like to thank the members of the Joint Integrated Avionics Working Group (JIAWG) Test and Maintenance Bus committee. Their ongoing design effort played a pivotal role in the development of the behavioral VHDL sequential circuit model. Further thanks go to my research's sponsors, Capt Jack Strauss and Nelson Estes. Without their efforts, this this work would surely not have begun. Finally, I would like to thank my wife, Wendy Haas. Without her, surely the trials and tribulations "affectionately" called AFIT would not have been bearable; BFNO.

Richard L. Miller

Table of Contents

List of Figures

Abstract

There exists an acute need for a methodology by which a circuit can be designed and validated against its specification before the circuit is fabricated. This thesis presents a merger of the specification and design capabilities of the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) with a verification method in order to solve the design and validation problem of sequential circuits. Currently, there is no methodology in-use, beyond exhaustive circuit simulation, which verifies the equivalence of an electronic system either to its specifications or to another electronic system. This problem is keenly apparent in the Air Force's Advanced Tactical Fighter program, the Navy's A-12 Fleet Defense Fighter, and the Army's LHX Attack Helicopter program. Congress has mandated that these vehicles utilize a common avionics architecture incorporating interchangeable, re-usable system modules. This interchangeability allows a tremendous amount of system flexibility; each air vehicle's avionics suite can be tailored from a common architecture into an integrated package that specifically meets the vehicle's combat mission. But, reusability depends on the equivalence of the system modules which, in this case, will be manufactured by three separate vendors working from a common system and module design specification. Currently, their module equivalence can be shown *only by an expensive and exhaustive simulation.* This thesis presents an alternative solution for sequential circuit development based upon the concepts of state equivalence. VHDL provides the capabilities of specification, design, and simulation for behavioral and structural electronic systems. Because of these features, it has been embraced by the DoD and has been designated as an IEEE standard. Further, a verification method exists whereby structural circuit descriptions can be tested for equivalence. This thesis reports on a proposed behavioral VHDL model for sequential circuit specification, a structural VHDL model for circuit description, and a software environment developed from UC Berkeley's VERIF software in order to accept both VHDL models and perform equivalence validation on the circuits these VHDL models describe.

Specification and Equivalence Verification

of

Sequential Circuits via VHDL

1 Introduction

There exists an acute need for a methodology by which a circuit can be designed and validated against its specification before the circuit is fabricated. The Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) provides the capabilities of specification, design, and simulation for electronic systems modeled either behaviorally or structurally. Further, verification methods exist whereby structural and behavioral circuit descriptions can be tested for equivalence. This thesis presents a merger of the specification and design capabilities of the VHDL with these verification methods in order to solve the design and verification problem of sequential circuits. The products are a behavioral VHDL sequential circuit model for sequential circuit specification, a structural VHDL model for sequential circuit design, and a method for comparing two circuits described using these VHDL models in order to demonstrate circuit equivalence. This chapter states the present problem, lays down the solution's scope and approach, discusses expected benefits, and outlines the thesis presentation.

1.1 Problem Statement

Currently, there is no methodology in-use, beyond exhaustive circuit simulation, which permits the equivalence verification of an electronic system either to its specifications or to another electronic system provided by a different vendor (1). This problem is keenly apparent in the Air Force's Advanced Tactical Fighter program, the Navy's A-12 Fleet Defense Fighter, and the Army's LHX Attack Helicopter program. Congress has mandated that these air vehicles utilize a common avionics architecture incorporating interchangeable system modules (2). This interchangeability allows a tremendous amount of system flexibility; each air vehicle's avionics suite can be tailored into an integrated package that specifically meets the vehicle's combat mission. But, interchangeability depends on the equivalence of the system modules which, in this case, will be manufactured by three separate vendors working from a common system and module design specification. Currently, module equivalence can be shown *only by an expensive and exhaustive simulation.*

The need for a verification methodology is also apparent in today's Computer Aided Design (CAD) environment. Design error detection and correction, which often occur late in a digital circuit's design phase, cause unexpected, often time-consuming delays in the circuit's production. By one account, 80% of all errors found during tests of the manufactured circuits are directly traceable to specification errors that cause deviations from the original design intent (1). Presently, these errors are detected by simulating the digital circuit's model prior to fabrication or by testing the manufactured circuit (1). Clearly, there exists an acute need for a methodology by which a module or a circuit can be designed and verified equivalent to its specification, another similar module, or a similar circuit.

1.2 Background

In order to reduce errors in designs and in turn reduce costs and manufacturing time, several different techniques have been employed within academia to prove that a digital circuit is either equivalent to its specification or that two digital circuits are equivalent to each other *without circuit simulation and before the circuit is fabricated.* While the most successful techniques have been applied to combinational logic circuits, several techniques for sequential logic circuits involving memory devices have recently been explored. Additionally, several CAD languages have been developed which attempt to address specific needs of sequential circuit design. None of these techniques are production quality CAD tools. Chapter 2 reviews these techniques.

1.3 Scope

The objective of this research is to produce a method for comparing two sequential circuits described via VHDL in order to demonstrate circuit equivalence. The two circuits can be described at the behavioral and/or structural design levels. This objective can be divided into two goals.

The first goal is to determine the appropriate VHDL language constructs to permit succinct structural and behavioral modeling of a sequential circuit. The products of this goal are two VHDL-based models, one for behavioral specification and another for structural sequential circuit description. The intention being that both models will prove sufficient for use not only as contractual documents but also as design tools. The second goal is to apply verification techniques to sequential circuits which are portrayed using the behavioral and structural VHDL models. The product is a set of software tools for comparing two sequential circuits described via the VHDL models in order to prove or disprove circuit equivalence before actual circuit fabrication.

Merging the specification and design capabilities of the VHDL with circuit verification methods solves the specification, design, and verification problem of sequential circuits.

1.4 Approach

After studying the features of description languages which can be used for sequential circuit modeling, similar language structures will be identified in the VHDL. Using these language structures, sequential circuits will be modeled and simulated in the VHDL at the behavioral and structural design levels to ensure that the VHDL models accurately portray the function of the desired sequential circuits. Further, sample system specifications will be reviewed to determine what additional specifications can be incorporated into a VHDL model. Finally, once behavioral and structural VHDL models have been developed and determined appropriate for modeling sequential circuits, a verification technique will be applied to the VHDL circuit models. This last step, in order to be a proof of concept, may require a restricted version of the VHDL models derived for sequential circuit specification and design.

1.5 Expected Gain

If this country is to remain technologically competitive in not only electronic system design but also CAD system development, the current design process of design verification by expensive and exhaustive simulation must change. Clearly merging the specification and design capabilities of VHDL with circuit verification methods offers a quicker and cheaper alternative to simulation. Additionally, this methodology permits the DoD to improve the procurement process by providing to the vendor complete system or circuit specifications in VHDL rather than an ambiguous English text. The vendor can check a system or circuit design directly against the VHDL specification rather than indirectly against the vendor's interpretation of an English text specification.

1.6 Thesis Overview

This thesis is presented in six chapters. Chapter 2 shows the results of a literature review of past and ongoing verification efforts and outlines several state machine description languages.

Also, Chapter 2 provides a description of state machine modeling and verification. Finally, Chapter 2 presents key features of VHDL useful for sequential circuit design and details the algorithms employed by the chosen verification software. Chapter 3 details the proposed VHDL models for state machine modeling. Chapter 4 explains the manner in which the chosen verification tool was adapted to VHDL. Chapter 5 presents sequential circuit designs utilizing the models of Chapter 3 and their verification results. Finally, Chapter 6 concludes with an overall summary and recommendations for future research efforts.

2 Background

This chapter provides background information concerning sequential circuits and presents the solution for this research. It is divided into three sections. The first describes sequential circuits; the second reviews past and ongoing verification efforts; and the third presents the solution to the stated problem of design verification.

2.1 Sequential Circuits

A sequential circuit is a circuit which, when given a set of inputs, produces an output which is a function not only of the inputs but also of an internally stored state of the circuit (3). Further, the state of the sequential circuit may change given a set of inputs and present state condition. A state machine (or finite state machine) is an abstract model used to describe a sequential circuit. Mathematically, simple state machines may be represented as (4):

A set of states represented by Q,

A finite set of input symbols, I,

A finite set of output symbols, Z ,

A mapping δ representing I x Q into Q, also known as a next state function, and,

A function ω representing I x Q onto Z, (for the Mealy machine) or

a function ω representing Q onto Z (for the Moore machine) also known as an output function.

State machines are pictured using directed graphs. This representation is called a state transition graph and is depicted in Figure 2.1. Additional state machine terms are defined as follows (3):

A state is a vector comprised of bits which may take on the values zero (0), one (1), or don't care (x). The number of bits in a state vector equals the number of memory devices in the state machine.

A minterm is a state vector which contains only the values one and zero.

One state covers another state if bits of one state are either equal to the corresponding bits of the second state or the bits of the first state are Don't Cares (x).

Transitions occur when the function δ produces a new state for the machine. Transitions are also called edges.

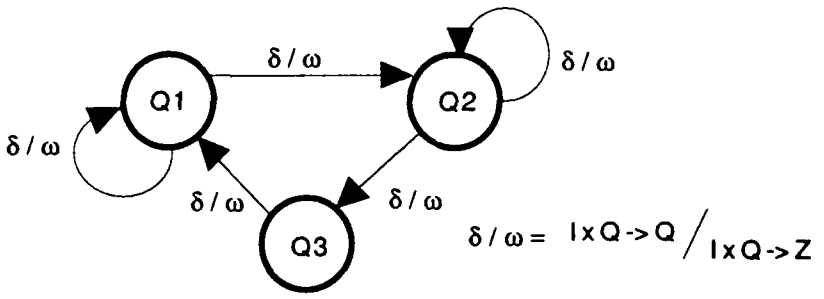
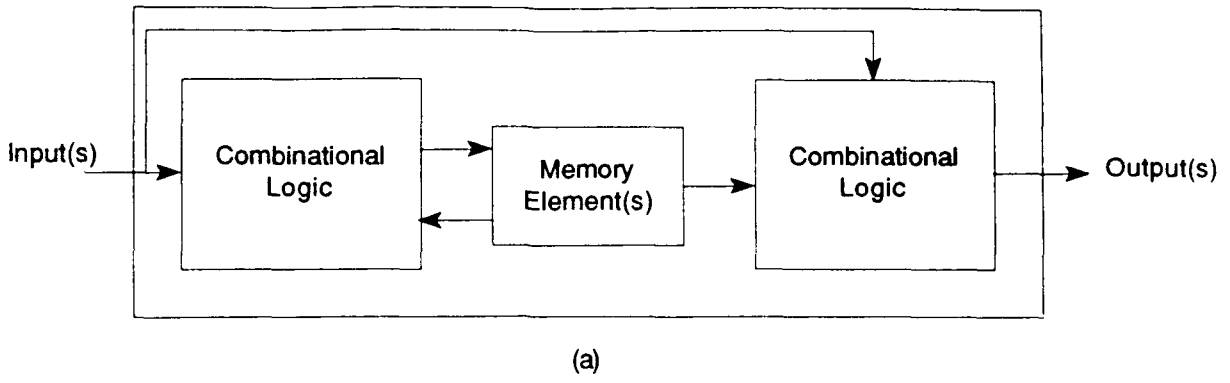


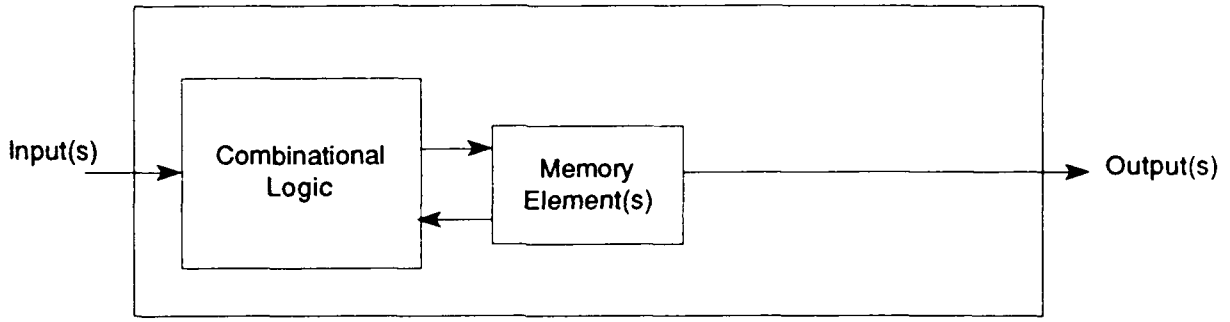
Figure 2.1. State Transition Graph.

The state machine of Figure 2.1 represents a Mealy machine. A Mealy machine is non-deterministic in that its output(s) depend not only on present state information but also its input(s). A Moore machine is deterministic in that its output(s) are dependent only on the present state of the machine. Figure 2.2 portrays this difference between machines. Figure 2.2(a) is a non-deterministic Mealy machine; Figure2.2(b) is a deterministic Moore. The terms state machine and sequential circuit will be used interchangeably throughout this thesis.

Additional types of state machines are hierarchical machines; and concurrent (hybrid) state machines. The hierarchical machines contains states in which additional state machines are nested, as depicted in Figure 2.3; hybrid machines contain multiple "processes" active within each state.



(a)



(b)

Figure 2.2. State Machines.

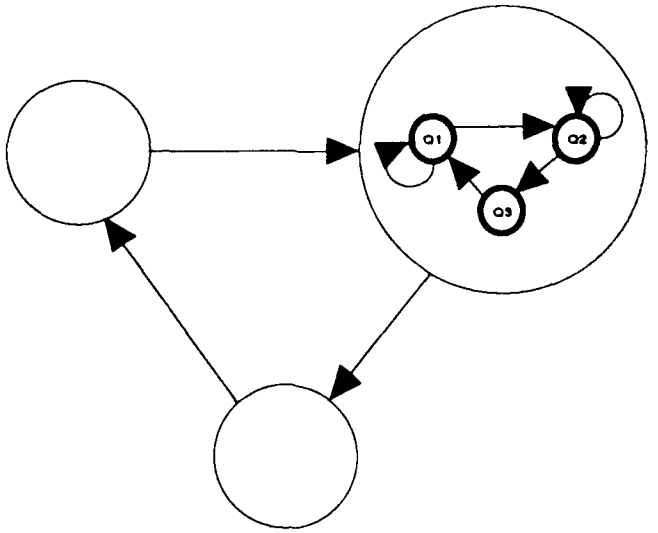


Figure 2.3. Hierarchical State Machines.

2.2 Verification

In order to reduce errors in designs and in turn reduce costs and manufacturing time, several different techniques have been employed to prove that a digital circuit is either equivalent to its specification or that two digital circuits are equivalent to each other *without circuit simulation and before the circuit is fabricated.* While the most successful techniques have been applied to combinational logic circuits, several techniques for sequential logic circuits involving memory devices have recently been explored. This section reviews past and ongoing verification research. The first section presents this work and its results; the second provides a more in depth explanation of selected verification methodologies.

2.2.1 Verification Efforts

Verification can be decomposed into two divergent methodologies: one attempts to prove the equivalence of a circuit to its specification through the application of formal mathematical methods to both the circuit and its specification; the other attempts to prove equivalence via searches or manipulations of alternate representations of the circuit's design space. Examples of both methods and their results are presented here. In regards to their results, one author's remarks are most telling: "The major stumbling block to formal verification methods is SKEPTICISM" (5).

Formal verification methods have already seen success in verifying the equivalence of various circuits allowing them to be fabricated prior to simulation (5). The British VIPER microprocessor was completly verified using the predicate-logic based, High Order Language (HOL) environment which is now available in the public domain. Produced at Cambridge University, the VIPER chip was extensively tested between specification levels. With exhaustive simulation only at the lower level portions of the design, no errors were found in the chip after the first fabrication. Similarly, Cambridge University developed TAMARACK, a processor similar to a pdp-11, which functioned correctly after the first fabrication with no pre-fab simulation.

Several Computer Aided Design (CAD) tools are available to aid in formal verification (5). HOL, mentioned above and now in the public domain, comes with its own attached functional language. LAMBDA, a commercially available product, also uses predicate logic, but includes a schematic capture feature. Additionally, it maintains specifications and tasks through logic decomposition.

Other methods of verification which lend themselves toward sequential circuit verification are under investigation. Recent work at UC Berkeley has developed two methods for verification: algorithmic path tracing within the sequential circuit's state transition graph and a state transition graph enumeration method which can be suported by interactive simulation(6, 7). No designs verified by these two methods have been reported fabricated at this time in the literature.

2.2.2 Verification Methods

Three verification methods are presented: symbolic logic, temporal logic, and state transition graph enumeration equivalence. Symbolic logic methods are used for predicate logic systems such as HOL. The verification method choosen for this research is similar in concept to the state transition graph method and is presented in Section 2.3.2.

2.2.2.1 Symbolic Logic Verification

Simply put, proof of equivalence via symbolic logic involves proving the equivalence of the boolean equations that describe the state machine to either its specification's boolean equations or to the boolean equations that describe another state machine (8). This technique is analogous to the trigonometry problem of proving the equivalence of the two sides of a trigonometric equation, such as $\tan(2AB) = \sin(A)\cos(B) + \cos(B)\sin(A)$, by the proper application of trigonometric identities.

Symbolic logic verification is hierarchical in nature (8). This means that it can be used to check chip, board, or system equivalence. The key to its use is that each chip, board, or system

level design must be representable via symbolic logic (8). Variables, constants, boolean, and arithmetic operators are permitted within the representation. The following example of a simple set/reset flip-flop constructed of nand gates (taken from (8)) is presented to demonstrate features of symbolic logic verification. In the example, the symbolic logic conditional operator notation has been simplified to facilitate reading; further information is available in (8).

The desired set/reset flip-flop functions are such that, when SET is false (logically a 0) and RESET is true (a logic value of 1), the output, Q, becomes true. Additionally, when SET is true and RESET is false, the output becomes false; and, when both SET and RESET lines are true, there is no change in the output. This functionality is symbolically specified as:

```
if not(SET) and RESET    then Q <- 1;
if SET and not(RESET)    then Q <- 0;
if SET and RESET         then Q <- Q;
```

The symbolic specification points out a key feature of symbolic logic verification. The specification does not dictate nor imply the implementation of the design; it simply specifies the functionality of the system. Figure 2.4 shows a typical implementation of the set/reset flip-flop.

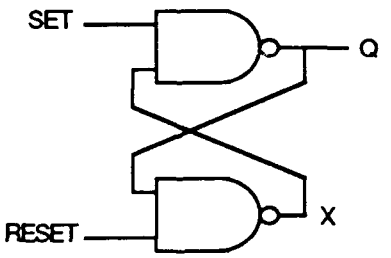


Figure 2.4. Typical Set/Reset Flip-Flop Implementation.

Translated to symbolic logic, the design is:

```
Q <- not(SET and X);
X <- not(RESET and Q);
```

In this representation, there is a total dependence of the design to the equations. A different design that implements the same functions would have different equations. To prove the equivalence of these two representations, verification proceeds by manipulating the logic equations to prove that the circuit description equations are equivalent to the specification equations. Continuing with the example, the verification steps are:

```
Q <- not(SET and X);

step1:
Q <- not(SET and not(RESET and Q)); Substitution for X

step2:
Q <- not(SET) or (RESET and Q);      DeMorgan's law

step3:
if not(SET) then Q <- 1 or (RESET and Q); Expansion
if SET then Q <- 0 or (RESET and Q);

step4:
if not(SET) then Q <- 1;                  Boolean Reduction
if SET then Q <- RESET and Q;

step5:
if not(SET) then Q <- 1;                  Expansion
if SET and RESET then Q <- (1 and Q);
if SET and not(RESET) then Q <- (0 and Q);

step6:
if not(SET) then Q <- 1;                  Boolean Reduction
if SET and RESET then Q <- Q
if SET and not(RESET) then Q <- 0;
```

At this point, the verification is complete; the equations derived from the original circuit description are equivalent to the specification. In other words, the circuit correctly implements the desired function.

This example shows that symbolic logic verification is a viable approach; but, it is not as simple as it seems. Although artificial intelligence programming techniques (in the computer languages Prolog or Lisp (1, 9)) are often employed to manipulate the symbolic equations, the verification process is not completely automatic. Often, the verification program requires considerable assistance from the designer to choose the correct theorem, substitution, expansion, or reduction to use in the next step (10). Additionally, several verification systems have been capable of verifying circuits only at the gate level (9). Because of these drawbacks, symbolic logic verification has been found to be effective on small sequential circuits utilizing four to six latches (7, 9).

2.2.2.2 Temporal Logic Verification

Although similar in approach to symbolic logic, temporal logic adds the important notion of time to hardware descriptions (11). Figure 2.5 graphically shows a waveform that can be expressed in temporal logic as $(\uparrow X, \downarrow X)^2$. This equation indicates that the signal X rises and falls twice during the time period of interest. Additionally, two key operators are always, ■, and sometimes, ◆. As an example, the equation $\blacksquare(Y = \neg(X))$ indicates that Y is always the not of X; while the equation $\blacklozenge(Y = \neg(X))$ indicates that Y is sometimes equal to the not of X. These constructs along with conventional logic operators permit reasoning about signals over time (11).

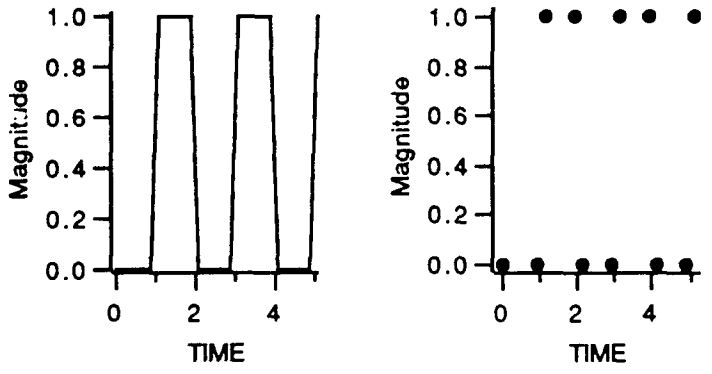


Figure 2.5 Continuous and Temporal Logic Waveform Representations.

Some derivatives of temporal logic also include the concepts of fairness, safety, and liveliness (10, 12). Fairness is a feature of interactive verification whereby the user enters a set of constraints that will be valid infinitely often along some path within the sequential circuit (10). An example would be a constraint which specifies that a process which continually requests access to a hardware resource, such as memory, will eventually be granted access to that hardware resource. The properties of safety and liveliness are properties specified by the user to be checked against the circuit's description (12); they can be thought of as further specifications beyond the functional specification. Safety properties specify that nothing wrong happens in the circuit; while liveliness properties specify good circuit actions. The concepts of "wrong" and "good" are relative to the specific circuit design.

Once the circuit has been described in temporal logic, several techniques are available to prove its equivalence to a specification or to another circuit expressed in temporal logic. As in symbolic logic verification, each method involves the manipulation of the logic equations until equivalence is shown. An important difference lies, though, in the equations' manipulation methodology and in the source of the temporal logic equations. Early verification efforts involved hand-generated temporal logic descriptions at the circuit specification level and at the gate level. Equivalence was then proven by direct manipulation of the temporal logic equations or their equivalent binary decision diagrams (10). Binary Decision Diagrams (BDD) are acyclic graph representations of boolean functions which provide a canonical form of the temporal logic functions (13). The circuits are shown equivalent if and only if their BDDs are equivalent.

Recently, significant work has been performed to automatically develop the temporal logic equations directly from a high level programming language or from a gate level design (13). The Compositional State Machine Language (CSML) allows for a hierarchical definition and interconnection of modules, compilation of the design into functional PLAs or PALs, and extraction of the equivalent temporal logic description (14). The automatically generated

equations can then be verified with the specification descriptions or against the gate level design (10). CSML provides high-level language features such as conditional and looping statements (if and while). Additionally, CSML allows for the concurrency of hardware operation with a parallel construct which permits concurrent statement evaluation (14). As a drawback, circuits described in CSML must be synchronous and deterministic in nature; additionally, CSML supports only two-level logic (zeros and ones) (14). The first requirement mandates one clock signal within the circuit; the second narrows the design space to Moore state machine descriptions. While the third requirement neglects the capabilities of multi-level logic (zeros, ones, strong, weak, charge, pre-charge, high-impedance, etc) exploited in other languages such as the DoD's VHSIC Hardware Description Language (VHDL). Finally, unlike VHDL, the CSML software environment does not support digital simulation of the circuit design.

2.2.2.3 State Transition Graph Enumeration Equivalence.

Proof of equivalence by state enumeration can be accomplished several ways (7). One method involves extracting the state transition graphs (STG) from two sequential machine descriptions and then showing their equivalence by exclusive-oring the STGs. Another approach extracts a STG from the first sequential circuit, uses the STG to determine the circuit's output on-and-off sets, and simulates these on the second circuit; the circuits are equivalent if the second circuit's outputs match the first's. A final method enumerates not only the inputs but also the state information of the two sequential circuits; equivalence is shown by differentiating between these inputs and states. After a brief description of a STG, these approaches are presented in order.

Figure 2.6 shows a sample STG. The circles indicate states of the sequential machine; arrow-headed lines indicate transitions between states.

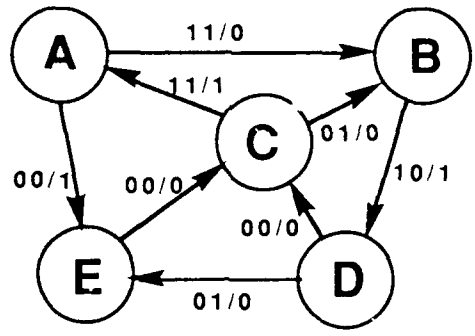


Figure 2.6. Sample State Transition Graph (STG).

The labels on the transitions, 11/0 for example, indicate input and output conditions during the transition; the 11 specifies two logic level 1 inputs and the 0 specifies one logic level 0 output. Each transition is triggered by a clocking pulse which is not shown on the graph. Although this STG represents a deterministic Moore sequential circuit, the STG verification techniques presented in this section are also capable of verifying non-deterministic Mealy circuits.

In the first verification method, an STG is enumerated for the first circuit by assuming output(s) for the first circuit and searching the circuit to determine the input set(s) required to produce the assumed output(s) (7). The STG generation process is repeated on the second circuit. These two circuit descriptions can be at the same, or at different, levels of design abstraction (7). Once the two STGs have been generated, a composite STG is created by exclusive-oring the two STGs together. If, in the third STG, no path exists between the starting state and any final states, the two sequential circuits are equivalent. If a path exists, the circuits are not equivalent.

The second technique employs the simulation of the first circuit's inputs on the second circuit. First, a STG is generated for the first circuit in the same manner as the previous technique. This STG is used to determine input stimuli to apply to the second circuit; don't care signal information is used to reduce the number of input sets required to test for equivalence. If, during the simulation, the second circuit's output(s) duplicate the first circuit's, the two are equivalent (7).

The final technique not only allows for the verification of discrete sequential circuits, but also permits verification of interacting sequential machines (6). Unlike the previous two STG methods, however, this technique requires that the input and the state space be explicitly enumerated (6). The approach to verification is that of checking for the equivalence of the reset/starting states of the two sequential circuits (6).

State transition graph verification methods have been shown viable on sequential circuits ranging from 15 to 250 latches with upwards of 10^{20} states (6). Both single or interacting, deterministic or non-deterministic sequential circuits can be verified (6, 7). Also, as opposed to exhaustive circuit simulation, the STG method verifies circuits in minutes rather than hours (6). Finally, it permits a versatile circuit description format at the gate, RTL, state table, or specification levels of design abstraction (7).

2.3 Solution

This section details the solution to the verification problem presented by this thesis. The solution is to marry sequential circuits described via a hardware description language to a known sequential circuit verification methodology.

2.3.1 Approach

The approach to the verification solution is broken down into two steps. The first step is to develop a behavioral and a structural modeling method for sequential circuits. Once these models have been developed, the verification software will be modified to accept as input sequential circuits described using these two models. The product is a method for comparing two sequential circuits described via the models in order to prove or disprove circuit equivalence before actual circuit fabrication. The VHSIC Hardware Description Language (VHDL) will be used to develop the models and the UC Berkeley verification software tools will be used for verification. First, the chosen verification tool set is described along with its theory of operation. Second, the

hardware description language is described with attention given to several key features of the language.

2.3.2 UC Berkeley Verification Tool Suite

The UC Berkeley verification tool suite consists of two tools: pre_verif and verif. The pre_verif software is a pre-processor for verif. Graphically, their interaction is shown in Figure 2.7. Using these two tools, the verification process is as follows. Pre_verif is invoked twice, once for each of the two structurally defined sequential circuits which are desired to be verified equivalent (or non-equivalent). This produces two input files for verif. In these two new files are the covers and some associated information from the two sequential circuits (the contents of which to be described in detail shortly). Then, verif is invoked and processes the two files to determine if the sequential circuits are equivalent. If so, verif exits with the statement

```
#MACHINES ARE THE SAME
```

otherwise the machines are not equivalent and verif exits with the statement

```
#MACHINES ARE DIFFERENT
```

followed by a set of input vectors (called the differentiating sequence) which, when applied as inputs to the sequential circuit, step the two machines through their states until the states of the two machines which exhibit different behaviors are reached.

The UC Berkeley pre_verif and verif software tools are available as source code from UC Berkeley for a DEC microVAX workstation. The tools are written in the programming language C and run on the VAX's ULTRIX operating system. ULTRIX is DEC's instantiation of UNIX for their microVAX workstations.

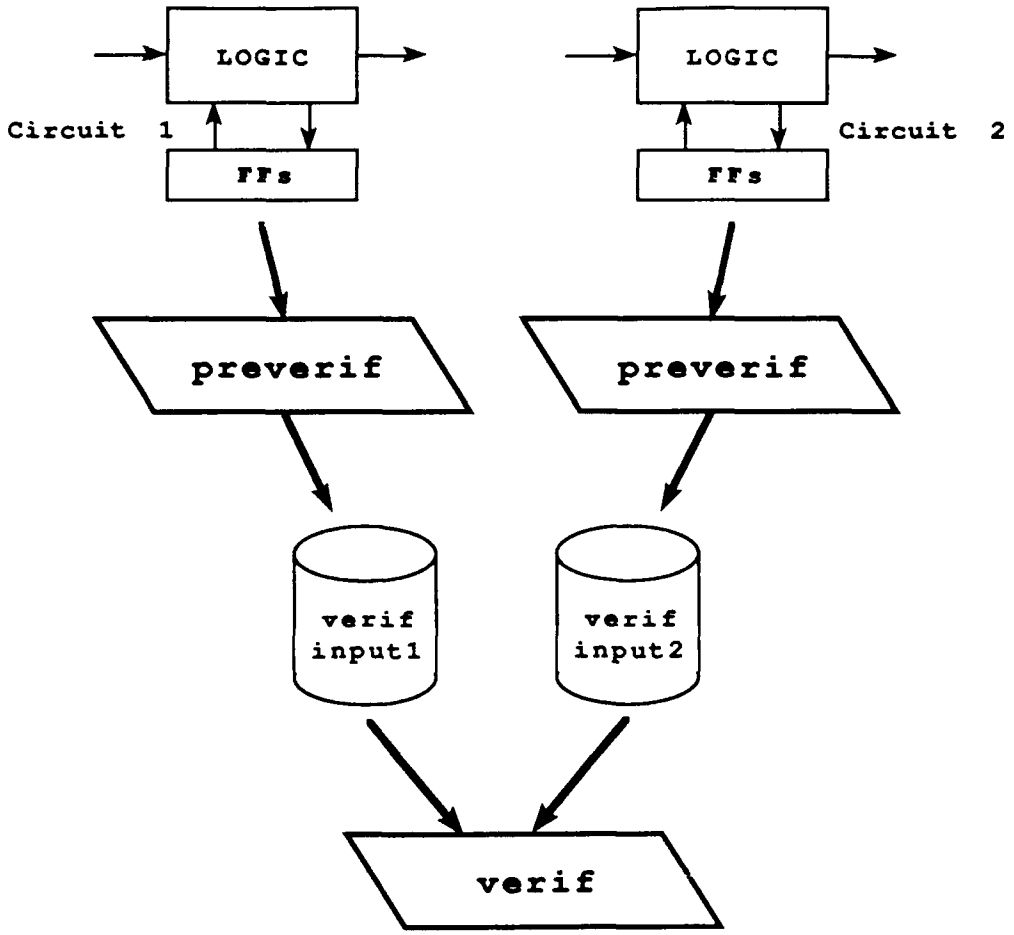


Figure 2.7. Verification Tool Suite.

2.3.2.1 Tool Methodology

This section presents an overview of the theory of operation of the two tools within the UC Berkeley Verification tool suite. Pre_verif is presented first followed by verif.

2.3.2.1.1 Pre_verif

Pre_verif is a preprocessor for verif which translates a sequential circuit described in the UC Berkeley netlist into a new file named "verif.input" which contains the output and next state cover information and associated minterm lists of the sequential circuit (6). For each output and next state signal within the state machine, pre_verif extracts input and next state covers which force the output(s) and next state signals to a logical one or zero. Additionally, pre_verif extracts a

list of input and next state signals which comprise the minterm variables for each output and next state signal. A final key piece of information extracted by pre_verif is the sequential circuit's initial or reset state. This information is stored in the file, verif.input, which is used as the input for verif.

2.3.2.1.2 Verif

Verif takes the cover, minterm, and initial state information generated by pre_verif from the two sequential circuits and performs the actual verification algorithms. Two algorithms contain the core of the verification process: Differentiate_States and New_Fanout_Edge. The main verification procedure is described in the following pseudo-code where M1 and M2 are represent sequential circuits one and two respectively (6).

```
verify_equivalence( M1, M2 )
{
      R1 = RESET State of M1;
      R2 = RESET State of M2;
      Flag = Differentiate_States ( R1, R2 );
      if ( Flag ) {
            /* Machines are different */
            Print_Differentiate_Sequence();
      }
      else {
            /* Machines are the same */
            Print_Valid_Invalid_State();
      }

}
```

The first algorithm, Differentiate_States, determines equivalence on a state by state basis. Starting with the initial state of each machine, verif sets an output of the first machine to a logical one. It applies the Path Oriented Decision Making (PODEM) algorithm to determine what input vector(s) are required to produce a logical one on machine one's output. Next, verif sets the corresponding output of machine two to a logical zero. Applying PODEM to the second machine, verif derives a second set of input vectors which produce a logical zero on machine two's output.

These two sets of input vectors are compared; if a vector is common to both sets of input vectors, the two machines are not equivalent and the verification process terminates.

Should the two sets of input vectors not have any vectors in common, Differentiate_States reverses the above process. The output of the first machine is set to a logical zero and the output of the second machine is set to a logical one. Sets of input vector(s) are derived for each machine and, as before, are compared for similar vectors. If a vector is common between the two sets, the two machines are not equivalent and the verification process terminates.

If the two states do not fail the input vector tests, then next states within the sequential circuits are calculated via the New_Fanout_Edge routine and the Differentiate_States routine is repeated recursively. This process follows paths within the state transition graphs which represent the two machines terminating a path only if the path doubles back on itself or if the inital state is reached. The path tracing is depicted in Figure 2.8. Because each verification starts at the intial state of each machine, the UC Berkeley reports that the verification problem is defined as the verification of the equivalence of the two sequential circuits' intial (or reset) states.

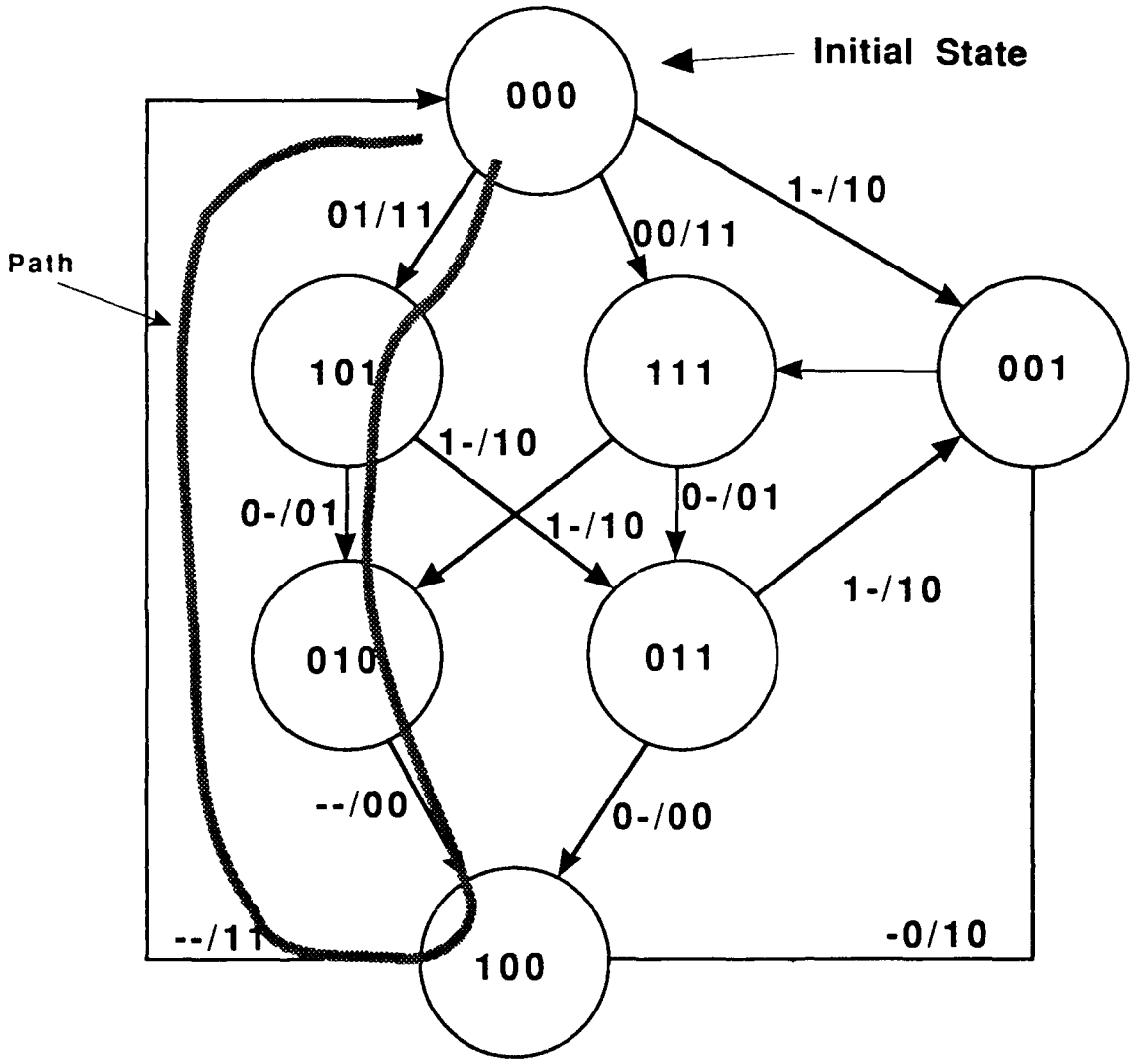


Figure 2.8. An Example STG (6).

These algorithms ares presented as the following pseudo-code where S1 and S2 are states of the two sequential circuits (6).

```
Differentiate_States( S1, S2 )
{
      if ( state pair (S1, S2) have already been examined )
            return ( Machines_Same );

      /* Find input combination which differentiates S1 & S2 */
      Flag = Find_Differentiating_Input ( S1, S2 );
      if ( Flag ) {
```

```
                /* such an input has been found */
                if ( inputs is not a don't care for S1 or S2 ) {
                        if ( output values are not don't cares ) {
                                Store input as part of differentiating
                                sequence;
                                if (no Don't Care sequences )
                                        return ( Machines_Different );
                        }
                }
        }
        Store State pair ( S1, S2 ) in current search path;

        /* Find input combination giving a new fanout edge */
        DecisionTree = NULL;
        S'1 = New_Fanout_Edge( S1 );
        S'2 = New_Fanout_Edge( S2 );

        while ( DecisionTree != NULL ) {
                Flag = Differentiate_States ( S'1, S'2 );
                if ( Flag ) {
                        /* such an input has been found */
                        if ( inputs is not a don't care for S1 or S2 ) {
                        if ( output values are not don't cares ) {
                                Store input as part of differentiating
                                sequence;
                                if (no Don't Care sequences )
                                        return ( Machines_Different );
                        }
                }

                S'1 = New_Fanout_Edge( S'1 );
                S'2 = New_Fanout_Edge( S'2 );
        }

        /* If this point is reached, machines are equal */
        return ( Machines_Same );
}
```

2.3.2.2 Verification Results

This verification method has shown promising results; it has been tested on various sequential circuits gathered from academia and industry sources with impressive results. Sample results are presented in Figure 2.9. The CPU times are for a VAX 11/8800 computer. The quoted units of time are s for seconds and m for minutes.

Circuit	#Inputs	#Outputs	#Valid States	#Edges	CPU Time
cse	7	7	16	141	.49s
sse	7	7	13	58	.18s
sand	11	9	32	183	1.34s
planet	7	19	48	142	.99s
sbc.4	33	24	54	19308	115s
sbc.1	16	1	65	1782	7.46s
scf	27	54	115	274	3.57s
tlc	3	5	400	2000	9.07s
mclc	11	6	35	917	5.17s
sbc.2	31	1	2040	2474094	563m
sbc.3	27	1	2764	1451108	140m

Figure 2.9. Verification Results (6).

2.3.3 VHDL

VHDL stands for the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language. Originally developed by the Department of Defense's Very High Speed Integrated Circuit (VHSIC) Program for use as a government standard (15), it has been adopted as an IEEE standard hardware description language (16). *This section explains not only why VHDL was* chosen as the specification and design language for this research but also describes some of VHDL's key features.

2.3.3.1 Why VHDL

As mentioned above, VHDL is the standard hardware description language for both the government and the IEEE. As such, it is intended to be used not only as a design language, but also it is intended as a specification language. VHDL provides language constructs capable of expressing both a structural design and a behavioral specification. As a standard, its use within this research effort promotes future acceptance of any products of this research work.

VHDL was not selected purely because it is both a DoD and an IEEE standard. The language was compared against other design languages which are used in academia to describe sequential circuits. From these comparisons, it became readily apparent that VHDL is quite

capable of describing sequential circuits both structurally and behaviorally. Two languages, State Machine Language (SML) and Compositional State Machine Language (CSML), provide many equivalent language constructs as VHDL (14, 17). Notably among them are conditional, loop, and concurrent statements. Additionally, the UC Berkeley circuit netlist format used by the verification software contains a subset of the information present in an equivalent circuit described in structural VHDL.

Further, work performed by both private industry and academia has shown that VHDL is capable of portraying sequential circuits. Two CAD tool developers are currently providing a sequential circuit VHDL code generation ability within their graphics-based CAD design environment (18, 19). The VHDL code which is produced by many designers to represent sequential circuits falls into two categories: separate procedure calls which represent each state and inline coding which groups the entire sequential circuit into one large monolithic block of code. Without delving into explicit explanations of the VHDL constructs, examples of these VHDL sequential circuit specification styles are represented in Figure 2.10.

```
Procedure_State_A( signal list );
Procedure_State_B( signal list );
Procedure_State_B( signal_list );
Procedure_State_C( signal_list );
Procedure_State_D( signal_list );
Procedure_State_E( signal_list );
Procedure_State_F( signal_list );
```

(a)

```
process ( NEXT_STATE ) begin

  STATE <= NEXT_STATE;

  if STATE = S0 then
     NEXT_STATE <= S1;
     Output1 <= '1';

  else if STATE = S1 then
     NEXT_STATE <= S2;
     Output2 <= '1';

  else if STATE = S2 then
     NEXT_STATE <= S1;
     Output1 <= '0';
     Output2 <= '0';

  end if;
  end if;
  end if;

end process;
```

(b)

Figure 2.10. Two Current VHDL Sequential Circuit Specification Styles.

Unfortunately, there are drawbacks in both code portions of Figure 2.10. The code of Figure 2.10(a) prevents the reader from developing any high level view of the sequential circuit. The contents of the procedure calls can be located in separate files or within the same file; whichever, anyone reading the code would be required to delve through its entirety in order to derive an understanding of the state transition graph. Likewise, the code of Figure 2.10(b) requires work on the reader's part to derive the state transition graph. Although 2.10(b) may appear to succinctly represent its sequential circuit, as the number of states and transitions grow, the code becomes all the more harder to interpret. Additionally, the VHDL code of Figure 2.10(b) cannot express separate, concurrent actions within a state. Existing design tools which produce VHDL code similar to Figure 2.10 use Computer Aided Design (CAD) based state machine editors which graphically portray the functionality represented by the VHDL code. Without this CAD aid, the

VHDL code becomes quite cryptic. One goal of this research is to develop a new behavioral VHDL sequential circuit modelling style which not only contains the same expressiveness as the existing VHDL sequential circuit models but also does not rely on any graphics-based CAD environment for improved understanding.

2.3.3.2 VHDL Features

This section details several of the key features of VHDL. It is not intended as a complete VHDL tutorial. Further information may be found in (15, 16). The language constructs used by this research can be separated into two categories: structural and behavioral.

2.3.3.2.1 Structural Features

Components within a VHDL description are comprised of two parts: the entity and the architectural body (16). From a "block box "perspective of the system, the entity is the black box which describes the components external interface. The entity provides the input and output wires (called ports) of the component and a capability of passing various values into the component. A typical entity is expressed as:

```
entity foo is
   generic (
             variable1 : integer;
             variable2 : real := 3.141592654
           );
   port (
             in_1  : in   bit := '1';
             in_2  : in   integer;
             out_1 : out  bit_vector (2 downto 0 )
         );
end foo;
```

This component has three unidirectional "wires:" in_1 and in_2 going into the black box and out_1 coming out. Of these, in_1 and in_2 are single wires while out_1 is a bundle, or bus, of 3 wires. Two variables, called generics, pass integer and real number information into the black box.

Additionally, one of the wires and one of the generics are initialized. Additional entity features may be found in (16).

Once entities have defined the component's black box external interface, netlists of components may be constructed representing the operation of a particular component. For example, the half adder of Figure 2.11 can be constructed of 3-input AND gates and one 4-input OR gate:

```
g1 : AND
      port map ( Anot, Bnot, C, g1out );
g2 : AND
      port map ( Anot, B, Cnot, g2out );
g3 : AND
      port map ( A, Bnot, Cnot, g3out );
g4 : AND
      port map ( A, B, C, g4out );
g5 : OR
      port map ( g1out, g2out, g3out, g4out, SUM );
```

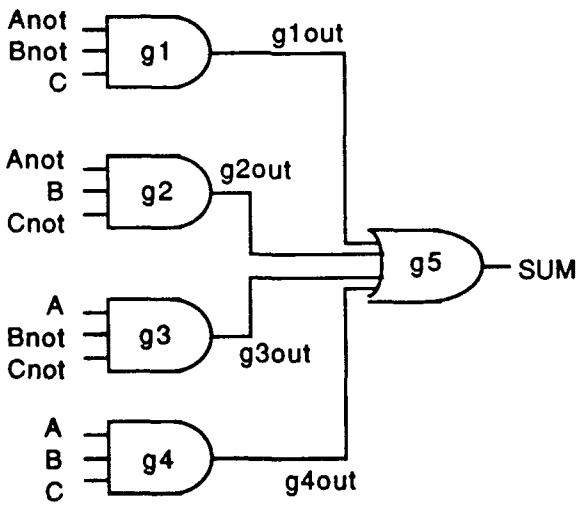


Figure 2.11. Half Adder.

For this example, the negated signals are assumed available at the entity's ports. This netlist of components representing the internal, gate level functionality of the half adder are "placed inside" the black box entity via the architectural body construct (16). The architectural body has the form:

```
architecture STRUCTURAL of half_adder is

      DECLARATIVE  BLOCK

begin

g1 : AND
      port map ( Anot, Bnot, C, g1out );
g2 : AND
      port map ( Anot, B, Cnot, g2out );
g3 : AND
      port map ( A, Bnot, Cnot, g3out );
g4 : AND
      port map ( A, B, C, g4out );
g5 : OR
      port map ( g1out, g2out, g3out, g4out, SUM );

end STRUCTURAL;
```

The architectural body's declarative block is that portion in which all components and signals are declared before use. In this case, these declarations would consist of an AND gate, an OR gate, and the signals g1out, g2out, g3out, and g4out. It is blank here purely for brevity. Additional architectural body features may be found in (16).

A powerful feature of VHDL is that it is not limited to purely structural representations of circuits. A functionally equivalent behavior may be placed inside a black box entity's architecture as:

```
architecture BEHAVIORAL of half_adder is

begin

     SUM <=  ( Anot and Bnot and C ) or ( Anot and B and Cnot ) or
             ( A and Bnot and Cnot ) or ( A and B and C );

end BEHAVIORAL;
```

This architectural body is functionally equivalent to the structural one depicted above. The "wire" SUM is assigned the value of the boolean equation:

$$SUM = \overline{A}\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot\overline{C} + A\cdot\overline{B}\cdot\overline{C} + A\cdot B\cdot C$$

With this ability to model circuits structurally or behaviorally (or with a sprinkling of both), VHDL offers the ability to specify and design electronic systems over a broad spectrum of the design hierarchy as depicted in Figure 2.12 (15). Additional key features of behavioral VHDL used in this research effort will be presented in the following section. Additional information may be found in (15, 16, 20)

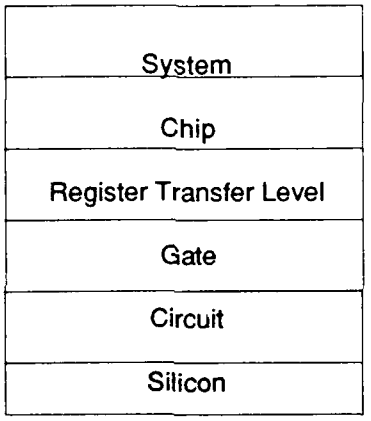


Figure 2.12. Design Hierarchy (15).

2.3.3.2.2 Behavioral Features

VHDL offers a wide range of constructs for describing the functionality of a component in a behavioral fashion. Of importance to this research effort are its ability to model actions which might take place concurrently with other actions within the component's architectural body; two of these constructs are VHDL's process and block statements (16). Additionally, in order to determine a signal 's (or "wire's") value when more than one language construct is attempting to place a value on that signal (called driving), VHDL provides resolution functions. Each will be

covered in this section. Further information regarding these constructs and others available within VHDL may be found in (15, 16, 20).

Representing the functionality of the entity foo presented above, an architectural body containing separate process constructs and and one block construct appears as:

```
architecture EXAMPLE of foo is

begin
   process ( in_1 )
      internal_variable : integer := variable1;
      begin
      -- statements go here.  See next process.
      end process;

   process( in_2 )
      begin
         out_1 <= "000";
      end process;

   A1: block ( in_1 = '1' )
      begin
         out_1 <= "111";
      end block A1;

end architecture EXAMPLE;
```

In this example, each construct functions independently of and concurrently with the others inside the architectural body. Just as for the architectural body, both the process and block construct have a declarative block and a statement part; for this example the first process contains a locally declared variable, internal_variable, which is also initialized to the value of the generic passed in from the component foo's entity generic. Additionally, each construct possesses a feature which controls the operation of the construct. For the process statement; that feature is a set of signals contained within parenthesis after the reserved word process ((in_1) and (in_2) above). The process construct only operates when a signal within its sensitivity list changes; otherwise, it sits dormant within the architectural body. For the block construct, a guard statement (the (in_1 = '1') following the keyword block) signifies a signal GUARD which is set to true if, and only if, the contents of the guard statement are true. This implied signal GUARD may be used

within the block to control the block's operation. As an additional feature, process constructs may appear within block constructs; but not vice-versa. The procedure calls of Figure 2.10(a) may appear inside both process and block constructs. It is the former case which is one method currently used to model sequential circuits as discussed in 2.2.3.1.

Within the example above, the signal out_1 is driven by one of the process and by one of the block constructs. Both values cannot be present on the signal (or wire) at the same time. VHDL provides a resolution function to resolve multiple signal drivers (16). Each time a value is assigned to a signal which has been declared with a resolution function, that function is called and it determines the proper value to place on the signal. A typical example of a user defined resolution function (which in this case implements a 'wired-or') is:

```
Function Resolve_Bits (il : bit_vector)
            return bit is

begin
      for I in il'Range loop
            if ( il(I) = '1' ) then RETURN il(I);
            end if;
      end loop;
      RETURN '0';
end;
```

Here, the resolution function checks each driver attempting to place a value on the signal. If any of the drivers are attempting to place a logical '1' on the wire, that value is placed on the wire; otherwise, a logical '0' is placed on the wire.

The process, block, and resolution function features of VHDL play a key role in the development of the behavioral VHDL sequential circuit model discussed in the next chapter. Further information regarding these constructs and others available within VHDL may be found in (15, 16, 20).

3 Sequential Circuit Modeling via VHDL

In this chapter, the VHDL models which are used in this thesis effort to support sequential circuit specification and design are described. Two models were developed; one for behavioral specification and another for structural design. These two models are capable of supporting a wide range of sequential circuit types. These types support both synchronous or asynchronous operation. Because this thesis effort is intended to demonstrate equivalence of sequential circuits described via VHDL, the behavioral and structural models employ specific VHDL constructs chosen to facilitate the equivalence demonstration. Hopefully, future revisions of the validation software will increase the language coverage to full VHDL 1076. The model restrictions are explained as each model is presented. Additionally, the behavioral model is geared towards clear specification of a design. Although upon first examination of the behavioral model this may appear to sacrifice succinctness for verbosity, the underlying requirement of the behavioral model is to be an easily readable, comprehendible, and stand alone specification. Each model is presented in the same fashion; first the model's concept is explained followed by an example.

3.1 EIA Conventions

To provide a degree of standardization between the behavioral and structural elements of a design -- and, for that matter, between entity and architectural revisions of the same design -- the EIA Commercial Component Model Specification SP-2229 was adopted as a standard convention. This standard specifies certain design conventions and includes a seven-level logic value family with supporting resolution, operator, and overloaded operator functions. Although this standard is intended to define the design's contents and acceptance criteria for commercially manufactured components, it is quite applicable to this effort. Several deviations were made from

the standard, but they are explained and justified when presented. The complete EIA VHDL standard package is attached as Appendix A.

3.2 The Model's VHDL Entity

This section describes the VHDL entity proposed for sequential circuits. It is intended for use with both the behavioral and structural architectural bodies and is described here to present a complete picture of the sequential circuit's VHDL model.

3.2.1 Entity Concept

The entity utilized in this effort follows the EIA guidelines with one major exception. The EIA standard specifies that every signal into or out of an entity should be a scalar signal. This requirement is levied by the EIA in order to provide a one-to-one correspondence between an electronic component's signals and that same component's packaging pins. In this thesis effort, this requirement was relaxed in order to permit vectorized entity ports. This decision is not contrary to the EIA standard. Logic cells within a component's architecture could readily employ vectorized ports on their entities as long as the "off-component" ports are scalars. Further, this port mode choice is bolstered in that at least one commercial vendor, ZYCAD, utilizes vectorized ports in their logic cell family (21). Finally, allowing vectorized ports permits a more robust software interface to the verification software as shall be seen in Chapter 4. As a last comment, VHDL generics are permitted within the entity as witnessed in the following example, but are currently ignored by the validation software.

3.2.2 Entity Example

Following the guidelines described above, a typical VHDL entity can be constructed as in the following example:

```
use work.Sequential_Circuit_Package.all;
      -- This package makes the appropriate type and
      -- variable declarations required for the
      -- particular state machine.  See text below.

use EIA.BASICDEFS.all;
      -- The BASICDEFS package is presumed located
      -- in a VHDL design library sublibrary named EIA.


entity Sequential_Circuit is
            -- generic();

      port  (
            input_1   : in logic_mv  := 'X';
            out_1     : out logic_mv := 'X';
            out_2     : out logic_mv := 'X';
            out_3     : inout logic_mv := 'X';
            RESET     : in logic_mv  := 'X';
            clock     : in logic_mv  := 'X'
            );

end Sequential_Circuit;
```

The Sequential_Circuit_Package package referenced in the above example is used to enumerate state names, transition labels, constants, a transition resolution function, and any other variables, constants, functions, or procedures required by the sequential circuit's entity and/or architectural body. Although several examples are presented in this thesis, the Sequential_Circuit_package package is intended to be tailored to the specific application. Further information is provided in Section 3.3.2.1 regarding this package's contents including a sample package development.

3.3 VHDL Behavioral Architectural Body

This section describes the behavioral VHDL model proposed for specifying sequential circuits. First, the model's overall concept is described including code fragments which support its functions followed by an example.

3.3.1 Behavioral Model Concept

Figure 3.1 represents a simple sequential circuit: two possible states and one possible transition between the two states. The depicted sequential circuit is comprised of two components -- states and transitions.

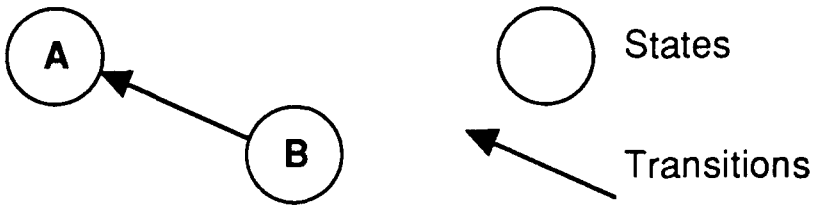


Figure 3.1. Simple Sequential Machine and Its Component Pieces.

Utilizing VHDL's block and process constructs, an architectural specification based on the state diagram may be decomposed into these same two components. Using this methodology, all transitions between states are handled by one VHDL process; state activities are handled by separate VHDL blocks, one per state. The partitioning of state to state transition information into one VHDL process permits a more succinct description the circuit's state machine diagram than those methods discussed in Section 2.2.3.1. This concept wil be further detailed in Section 3.3.1.1. Any additional actions, such as clock synchronization, global reset or set, test circuitry, etc, may also be included as concurrent VHDL blocks within the behavioral architecture. Using this method, the behavioral architectural body's skeleton form for the entity Sequential_Circuit is:

```
architecture behavioral_body of Sequential_Circuit is

      -- Declarative Block

begin
      process ( transition )
            begin
            end;
```

```
        FIRST:block ( Present_State = First_State )
              begin
              end;

        Second:block ( Present_State = Second_State )
              begin
              end;
        .
        .
        .
        nth_block:block ( Present_State = nth_State )
              begin
              end;

        process ( clock )
              begin
              end;

        RESET_BLOCK:process ( RESET )
              begin
              end;

end behavioral_body;
```

By using sensitivity lists on processes and guard statements on blocks, the processes and blocks function concurrently. The contents of the processes and blocks within the context of the behavioral model are described in the following sections.

This methodology was chosen in that it presents the design as an assemblage of smaller pieces -- basically states and transitions -- while at the same time each piece succinctly specifies its own actions. Transitions, states, and other concurrent actions are described as follows.

3.3.1.1 Transition Process

The transition process within the behavioral architectural specification is used not only to resolve all state-to-state transitions of the sequential circuit, but also to present in simple fashion a skeletal structure of the overall sequential circuit. The former is required for the machine to function; the latter succinctly presents the overall machine in a quite human-readable form. In terms of design specification, the ease in extracting the overall machine diagram is paramount – no CAD based software tool is required to graphically clarify the state machine design as required

in other sequential circuit specification styles. Separating the state-to-state transition from its respective state provides a clear method to provide a "snapshot" of the design. In other words, a designer can readily sketch the sequential circuit's state machine diagram from the transition process block.

Figure 3.2 shows a sample sequential machine. For this example, only state-to-state transitions are labeled; no output signal lines are present. Further, the existence of transition, Present_State, and Next_State signals, whose types are enumerated in the Sequential_Circuit_Package included in the entity description, are assumed. Visibility into this package is accomplished by the appropriate VHDL use statement at the architecture's entity. The types enumerated in the Sequential_Circuit_Package are:

```
Type Transition_Conditions is (
                    No_Transition,
                    goto_First_State,
                    goto_Second_State,
                    goto_Third_State
                    );

Type States is (
                    Unknown_State,
                    First_State,
                    Second_State,
                    Third_State
                    );
```

No_Transition and Unknown_State are provided for the situation where no transition occurs between states and for power-up when the machine is in an unknown, or uninitialized, state. With this information, the skeletal sequential machine can be readily constructed from the transition process.

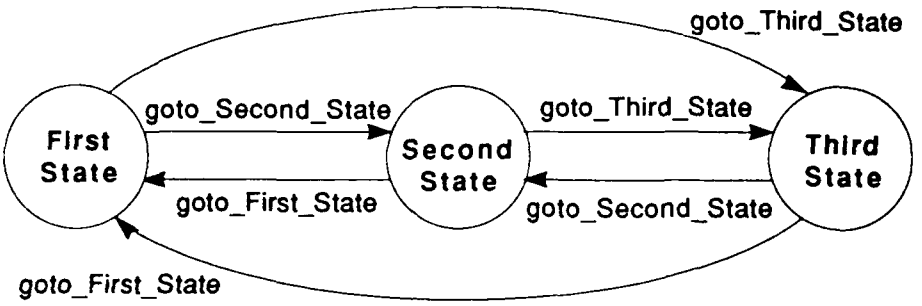


Figure 3.2. Skeletal State Machine.

A VHDL transition process representing this machine is :

```
process ( transition ) begin
      case Present_State is
      when First_State =>
            case transition is
                  when goto_second_State =>
                        Next_State <= Second_State;
                  when goto_Third_State =>
                        Next_State <= Third_State;
                  when others =>
            end case;

      when  Second_State =>
            case transition is
                  when goto_First_state =>
                        Next_State <= First_State;
                  when goto_Third_state =>
                        Next_State <= Third_State;
                  when others =>
            end case;

      when  Third_State =>
            case transition is
                  when goto_First_state =>
                        Next_State <= First_State;
                  when goto_Second_state =>
                        Next_State <= Second_State;
                  when others =>
            end case;

      when Unknown_State =>
            case transition is
                  when goto_First_state =>
                        Next_State <= First_State;
                  when others =>
            end case;

      end case;
end process;
```

Although this transition process reveals no information concerning the internal workings of the machine's states, it's apparent from this example that it clearly describes the overall machine diagram -- a very useful feature for creating a lucid design specification.

3.3.1.2 State Blocks

The individual states of the sequential circuit are represented by VHDL block constructs. For the simple Moore or Mealy sequential circuit, these blocks simply set the values of the output signal(s) and test for the transition condition(s) into another machine state. More complex designs may contain multiple concurrent activities represented as VHDL processes operating within the state's block. In the extreme, this behavioral model allows hierarchical state machines whereby entire state machines may be nested within state blocks.

To permit this type of operation two conditions must be met by the state block. For the first condition, each signal which is driven by more than one state block must be provided with two features. Each multiply-driven signal must have a bus resolution function. An example transition resolution function used throughout this research is:

```
Function Transition_Resolution (i1 : Transition_Conditions_vector)
            return Transition_Conditions is

begin
      for I in i1'Range loop
            RETURN i1(I);

      end loop;
      RETURN No_Transition;
end;
```

This function assumes that one, and only one, state block will be making an assignment to the Transition signal. This is guaranteed by requiring that any non-executing state must disconnect its signal driver from the Transition signal by the assignment of null (as shown in the state block example to follow).

The second condition to meet requires that each state block's operation be determined by the value of a guarded signal which checks the present state of the sequential circuit. The guard signal will be true only if the the circuit's present state is the same state represented by the block. Processes within the state block check this guard signal and execute only when the guard statement is true. Given this, a simple state example which assigns the value '0' to its output (represented by Machine_output) and transitions into a second state when the input signal is a "11" vector is:

```
FIRST:block (Present_State = First_State) begin

      process (GUARD, INPUT_signals) begin
            if GUARD then
                  Machine_output <= '0';
                  if INPUT_signals = "11" then
                        Transition <= goto_Second_State;
                  end if;
            else
                  transition <= null;
                  Machine_output <= null;
            end if;
      end process;

end block FIRST;
```

In this example, the state "operates" when its guard statement,

```
Present_State = First_State
```

is true. If false, the process assigns null to both transition and output signals.

More complex state blocks may include one or more concurrent process blocks; or, in the extreme, a state machine represented by its own transition process and state blocks. A multiple process state block could appear as:

```
READ_Instruction: block (Present_State = READ_Instruction_State) begin
      process (GUARD) begin
            if GUARD then
                  Control <= C9_C3;
            else
                  Control <= null;
            end if;
      end process;

      process (Clock)
            variable clock_count : integer := 0;
      begin
            if GUARD and negedge( clock ) then
                  clock_count := clock_count + 1;
                  if clock_count = 2 then
                        clock_count := 0;
                        Transition <= goto_IR_gets_DR_OP;
                  end if;
            else
                  Transition <= Null;
            end if;
      end process;

      end block READ_Instruction;
```

In this example, the first process (sensitive to the state's guard statement) simply assigns an output to the Control line. The second process (sensitive to the negative edge of the clock and checking the value of GUARD on each execution) counts two clock pulses before making the transition assignment out of the READ_Instruction state. Note that both processes assign null to the Control and Transition signals during the else clause of their conditional statements. This requirement is levied by the chosen transition resolution functions.

By permitting multiple processes or nested blocks within the state block, the behavioral model allows for a flexible design style -- hierarchical decomposition is easily achieved. But, one must be careful in using the model not to deviate from the model's primary intent: clear behavioral specification.

3.3.1.3 Additional Concurrent Actions

This section describes any additional concurrent blocks or processes that may be included in the behavioral architectural specification. This is not an all inclusive set of additional concurrent processes or blocks -- these are the essential elements to complete the proposed behavioral VHDL model. Although a designer could easily add any number of additional processes or blocks, succinctness and clarity should be maintained.

As stated earlier, this behavioral model supports both synchronous and asynchronous operation. This feature is accomplished by the following process. Operating concurrently with the transition process and state blocks, this process synchronizes the state transitions to the clock.

```
process (clock) begin
            if (clock = '1' and clock'event)
                  then Present_State <= Next_State;
            end if;
end process;
```

By removing this process from the sequential circuit's architectural body and changing the concurrent transition process' signal assignment statement from:

```
Next_State <= some_state;
```

which assumes that the variable Present_State is assigned in the clock process to:

```
Present_State <= some_state;
```

the sequential machine becomes asynchronous. Appendix B contains an asynchronous example.

Additionally, a reset or initialize capability can be provided in a like manner. Another process operating concurrently with the transition process and state blocks provides this global reset and initialize capability:

```
-- initialize and reset capability
RESET_BLOCK:block (RESET = '1') begin

      process (GUARD) begin
            if  GUARD then
                  Transition <= goto_First_State;
            else
                  Transition <= null;
            end if;
      end process;

end block RESET_BLOCK;
```

Again, the model's capabilities are up to the individual designer, these simple cases have been presented only as example.

3.3.2 Behavioral Model Example

While simple in concept, the VHDL behavioral model is capable of specifying many different types of sequential circuits: Moore, Mealy, "hierarchical" Moore and Mealy, and "hybrid" machines. The "hierarchical" machine implies state machines nested within states of the sequential circuit; the hybrid machine implies multiple concurrent processes within the states of the state machine. All these machines may be either synchronous or asynchronous. The following example is a synchronous control unit for a simple eight-instruction CPU. Chapter 5 and Appendix B contains further examples of various types of sequential machines designed using this behavioral model.

3.3.2.1 CPU Controller

The following example taken from (22) is a CPU controller intended to operate as a controller for an eight-instruction CPU. The CPU is to perform the following instructions:

LOAD X	transfer contents of memory location X into accumulator.
STORE X	transfer contents of accumulator to memory location X.
ADD X	Add contents of memory location X to contents of accumulator and store in accumulator.
AND X	Logical AND the contents of memory location X with the contents of accumulator and store in accumulator.
JUMP X	Unconditionally branch to the instruction stored in memory location X.
JUMPZ X	If the accumulator equals zero (as specified by the zero flag), branch to the instruction stored in memory location X.
COMP	Complement the accumulator's contents and store in the accumulator.
RSHIFT	Right-shift the contents of the accumulator and store in the accumulator.

Given this instruction set, Figure 3.3 presents a block diagram of the CPU as specified in (22). One hardware constraint not depicted in Figure 3.3 demands that all micro-operations require one clock cycle except for memory accesses which require two clock cycles. Finally, control signals must remain valid for the entirety of the micro-operation. The goal, then, is to design a sequential circuit which performs the control unit functionality as depicted in Figure 3.3.

The first step in the design process is to develop the control unit's entity description. This is accomplished simply by matching signals depicted in Figure 3.3 to ports on the entity. Additionally two assumed signals, reset and clock, which are not shown in the figure, are included in the control unit. The clock signal will be used to synchronize the sequential circuit's state transitions and control signals; reset will be used to initialize or force the control unit into a known initial state. Further, the reset signal will be developed as an asynchronous signal not dependent on the clock.

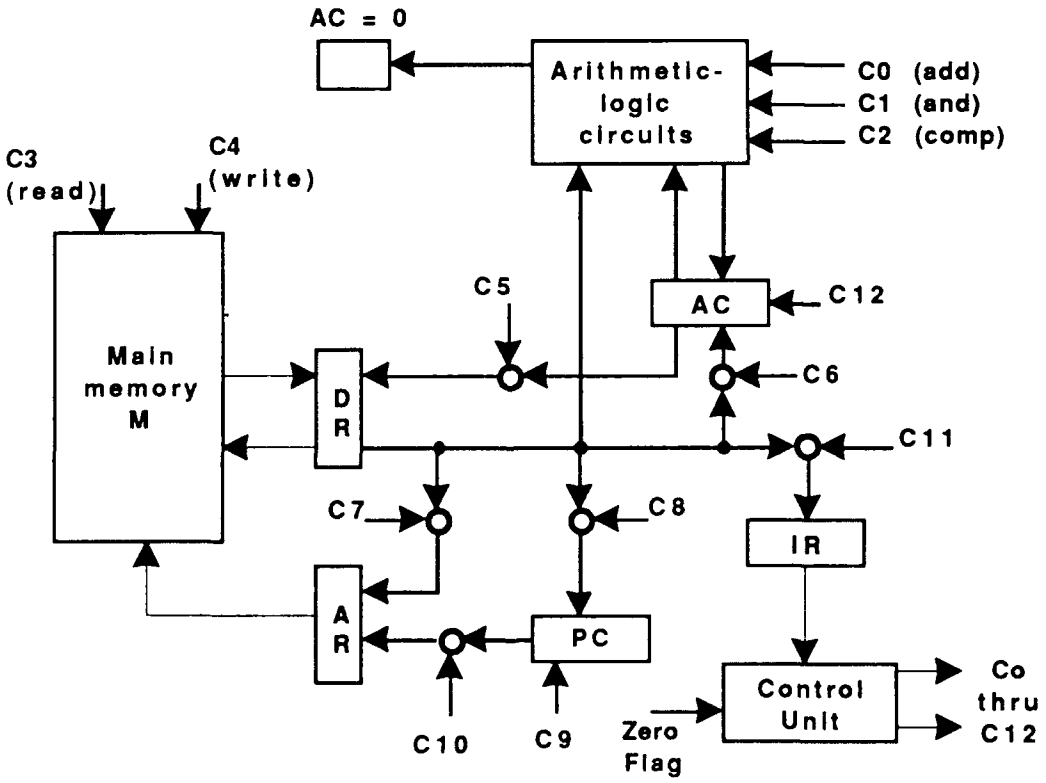


Figure 3.3. Block Diagram of the Eight-Instruction CPU (22).

The entity description is then:

```
use work.BASICDEFS.all;
      --  Again, BASICDEFS is the EIA's BASICDEF package.
use work.CPU_package.all;
      --  The CPU's Sequential_Circuit_Package.
entity CPU_CONTROLLER is
      -- generic ();
      port( instruction : in instructions := NOP ;
            CLOCK       : in logic_mv;
            RESET       : in logic_mv     ;
            ZERO_FLAG   : in logic_mv;
            Control_bus : out logic_mv_vector_bus (12 downto 0)
                        := "XXXXXXXXXXXXX"
            );
end;
```

The ports are defined as follows. Instruction is an enumerated set of instructions. The instructions type will be developed shortly in the Sequential_Circuit_Package named CPU_package. Clock, RESET, and ZERO_Flag are all multi-value logic scalar signals. Finally, Control_bus is a multi-value logic vector of signals representing the CPU's control signals C12, C11, ..., C0. Figure 3.4 shows the control signal to CPU micro operation relationship.

Control Signal	Micro-operation
C0	AC <- AC + DR
C1	AC <- AC AND DR
C2	AC <- NOT AC
C3	DR <- M(AR) = READ M
C4	M(AR) <- DR = WRITE M
C5	DR <- AC
C6	AC <- DR
C7	AR <- DR(ADR)
C8	PC <- DR(ADR)
C9	PC <- PC + 1
C10	AR <- PC
C11	IR <- DR(OP)
C12	RIGHT-SHIFT AC

where AC = Accumulator
DR = Data Register
AR = Address Register
PC = Program Counter
OP = Op Code
ADR = Address

Figure 3.4. Control Signal to CPU Micro-operation Relationship (22).

Next, a state diagram must be derived from the behavior of the sequential circuit. This behavior is shown in Figure 3.5 (22). It's important to note that at this point in the behavioral architecture's development, the design can be written several ways. One method produces a "hybrid" Moore sequential circuit where each micro-operation represents a state of the sequential circuit. An alternative method uses a nested state machine approach. Here, the top-level state machine would have two states: Fetch and Execute. These two states break the CPU's behavior into two distinct phases. Implemented in this manner, the Fetch and Execute states would contain their own state machines each issuing respective control signals synchronized with the proper clock cycles. As a final alternative, the behavior could be developed as a state machine possessing, at a minimum, eight states. Each state represents one member of the CPU's instruction set of LOAD, STORE, ADD, AND, JUMP, JUMPZ, COMP, and RSHIFT. The behavioral model presented in this thesis is quite capable of modeling the controller in these different abstractions of its behavior. For the sake of a simple example, however, the first method will be developed: each micro-operation will represent a state of the sequential circuit. Given this, a state machine diagram representing this design is shown in Figure 3.6. Not shown are each state's RESET transitions into the "AR <- PC state."

The Sequential_Circuit_Package can be developed directly from the state machine diagram. Named CPU_package for this example, this package enumerates the states, transition conditions, and instructions for the controller. Additionally, it declares ancillary constants and functions. The CPU_package, along with the entirety of the CPU controller's VHDL code is presented in Appendix B.

Once the CPU_package has been developed, the skeletal VHDL code which represents the CPU controller can be fleshed out from the clock and reset processes, a transition process, and eleven state blocks. The skeletal structure is:

```
architecture BEHAVIORAL of CPU_CONTROLLER is

begin

CLOCK_SYNCH:process (clock)
            begin
            end CLOCK_SYNCH;

RESET_BLOCK:process (RESET)
            begin
            end;

process (transition)
      begin
      end;

AR_gets_PC:block (Present_State = AR_gets_PC_State)
            begin
            end AR_gets_PC;

READ_Instruction:block (Present_State = READ_Instruction_State)
            begin
            end READ_Instruction;
      .
      .
      .

RIGHT_SHIF_AC:block (Present_State = RIGHT_SHIF_AC_State)
            begin
            end RIGHT_SHIF_AC;

end BEHAVIORAL;
```

The Clock_SYNCH and RESET_BLOCK processes are similar to those presented earlier in this chapter. For completeness, the[illegible]re included in Appendix B. Of prime importance, now, is the development of the transition

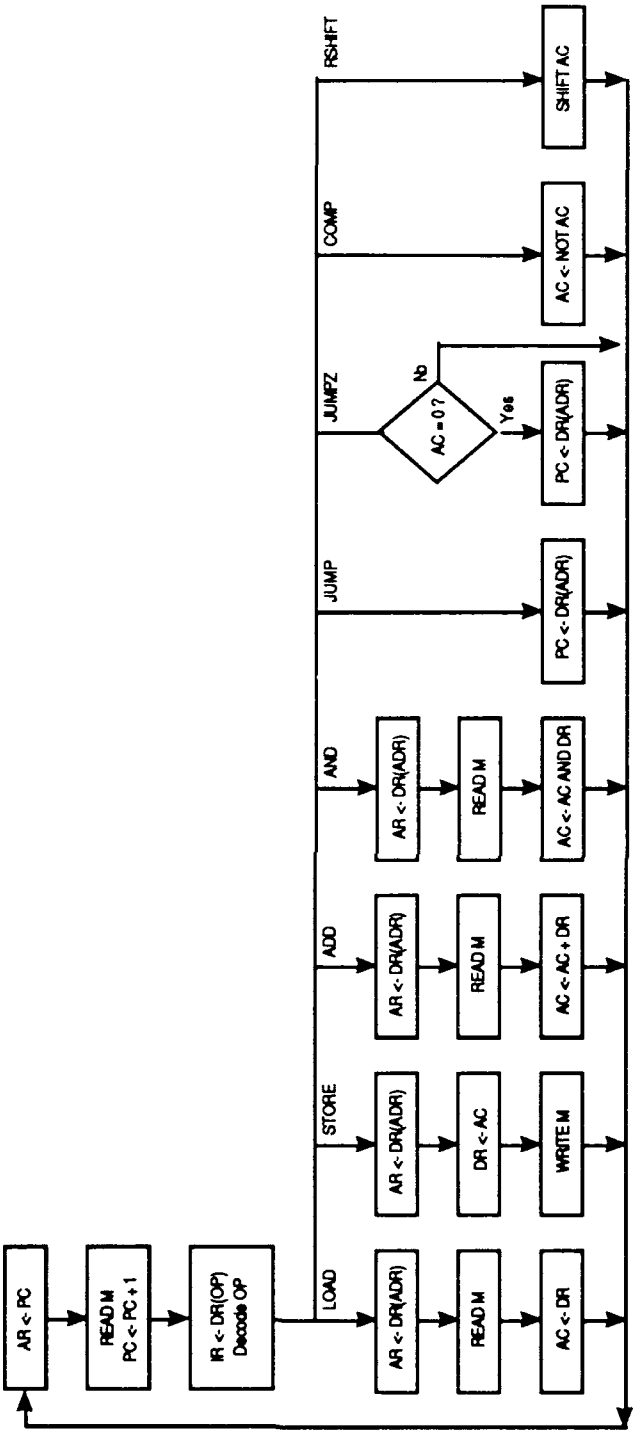


Figure 3.5. Controller's Behavioral Operation (22).

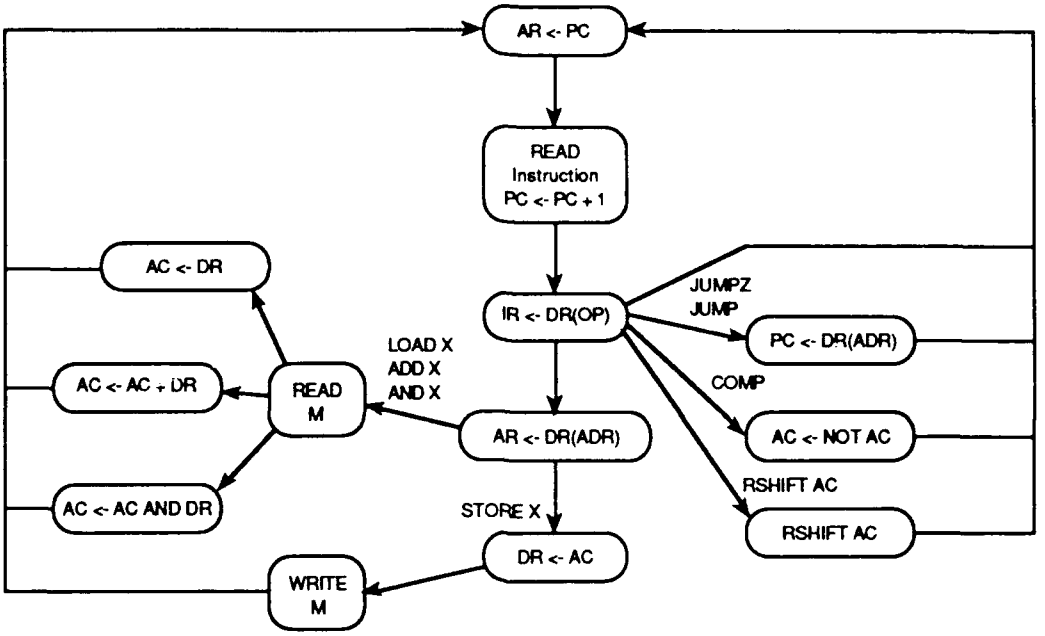


Figure 3.6. CPU Controller State Machine Diagram.

process. This process can be taken directly from the state machine diagram. Not labeled in the figure, the transitions are:

```
No_Transition,
goto_RESET,
goto_AC_gets_DR,
goto_AC_gets_AC_plus_DR,
goto_AC_gets_AC_and_DR,
goto_IR_gets_DR_OP,
goto_AR_gets_DR_ADR,
goto_AR_gets_PC,
goto_AC_gets_NOT_AC,
goto_RIGHT_SHIFT_AC,
goto_Write_M,
goto_READ_M,
goto_DR_gets_AC,
goto_READ_INSTRUCTION, and
goto_JUMP
```

Following the format described in Section 3.3.1.1 and recording the transitions from state to state, the transition process is of the form:

```
process ( Transition ) begin

      case Present_State is

            when AR_gets_PC_State =>
                  case Transition is
                        when goto_RESET =>
                              Next_State <= AR_gets_PC_State;
                        when others =>
                              Next_State <= READ_INSTRUCTION_State;
                  end case;

            when READ_INSTRUCTION_State =>
                  case Transition is
                        when goto_IR_gets_DR_OP =>
                              NEXT_STATE <= IR_gets_DR_OP_State;

                        when goto_RESET =>
                              Next_State <= AR_gets_PC_State;

                        when others =>
                  end case;

            when IR_gets_DR_OP_State =>
                  case Transition is
                        when goto_AR_gets_DR_ADR =>
                              NEXT_STATE <= AR_gets_DR_ADR_State;

                        when goto_AR_gets_PC =>
                              NEXT_STATE <= AR_gets_PC_State;

                        when goto_JUMP =>
                              NEXT_STATE <= JUMP_State;

                        when goto_AC_gets_NOT_AC=>
                              NEXT_STATE <= AC_gets_NOT_AC_State;

                        when goto_RIGHT_SHIFT_AC=>
                              NEXT_STATE <= RIGHT_SHIFT_AC_State;

                        when goto_RESET =>
                              Next_State <= AR_gets_PC_State;
                        when others =>
                  end case;

            .
            .     -- Remainder of transition process deleted for
            .     -- clarity's sake.  See Appendix B.


            when others =>
      end case;
end process;
```

Finally, the state blocks can be fleshed out. Two representative states, AR_gets_PC and READ_M, are presented to exemplify the controller's behavioral development. Again, the controller's complete VHDL code is presented in Appendix B along with a sample simulation report of its operation.

The first state, AR_gets_PC_State, sets C10 equal to '1' and all other control lines equal to a '0.' Additionally, transitions are made unconditionally from this state to either itself (the RESET condition) or into the READ_Instruction state. RESET is handled globally by the reset process; but, the Transition signal must be set by the AR_gets_PC_State. Furthermore, when not in the state, the drivers on the signals Transition and Control must be assigned null values. The VHDL code representing this state is:

```
-- AR_gets_PC state
AR_gets_PC: block (Present_State = AR_gets_PC_State) begin
      process (GUARD) begin
            if GUARD then
                  Control <= C10;  -- C10 defined in CPU_package
                  Transition <= goto_Read_Instruction;
            else
                  Transition <= null;
                  Control <= null;
            end if;
      end process;
      end block AR_gets_PC;
```

The READ_INSTRUCTION_State is slightly more complex and is a good example of the behavioral model's specification capabilities. As in the previous state example, the control signal must be set ; a straightforward process. Any memory access, however, must take two clock cycles to account for the slower nature of the CPU's memory. Because READ_INSTRUCTION_State accesses memory, this two clock cycle time delay is accomplished by an additional process within the state block that counts clock pulses (in this case negative edges) and only permits the transition signal to take place after two clock cycles have elapsed.

The function negedge() is defined in the EIA's BASICDEFS package included in Appendix A.

The READ_INSTRUCTION_State's VHDL code is:

```
-- READ_INSTRUCTION state
--    This state not only reads the instruction from memory,
--    but also increments the PC.
READ_Instruction: block (Present_State = READ_Instruction_State)
begin
      process (GUARD) begin
            if GUARD then
                  Control <= C9_C3;  -- Defined in CPU_package.
            else
                  Control <= null;
            end if;
      end process;

      process (Clock)
            variable clock_count : integer := 0;
      begin
            if GUARD and negedge( clock ) then
                  clock_count := clock_count + 1;
                  if clock_count = 2 then
                        clock_count := 0;
                        Transition <= goto_IR_gets_DR_OP;
                  end if;
            else
                  Transition <= Null;
            end if;
      end process;

end block READ_Instruction;
```

Following this methodology, the VHDL code for the remaining states is constructed in a similar manner. It is apparent from this example that this behavioral model not only permits a straight-forward construction method but also provides a clear presentation of the sequential circuit's behavior. The complete VHDL code and a sample simulation report for the CPU controller is in Appendix B.

3.4 VHDL Structural Architectural Body

This section describes the structural VHDL model for designing sequential circuits. First, the logic gate selection and their selection rationale are explained followed by an explaination of

the architectural body's structural layout and overview of allowed language constructs. Finally, an example using the defined logic gates and structural architecture is presented.

3.4.1 Structural Model Concept

The components chosen for the structural model are directly related to the capabilities of UC Berkeley's verification tools: Senum, Pre_Verif, and Verif. These programs' input format can be compared to a structural VHDL description in that their input file formats describe a netlist of components. Those components recognized by the UC Berkeley tools consist of inverters, ANDs, NANDs, ORs and NORs. Additionally, two flip-flops are supported: clocked and asynchronous D flip-flops. To serve as a proof of concept, then, the structural VHDL model was choosen to closely follow the UC Berkeley input format; entities have been described paralleling the recognized components. Additionally, simplistic architectures that mimic the component's functional behavior have been produced to support VHDL simulation. These architectures do not include propagation delay or other timing information; but "generic hooks" are provided for future work. Two sample components follow: an AND gate and a clocked D flip-flop; a complete list of components along with their VHDL code is provided in Appendix C. As described in Section 3.2.1, all entities use vectorized entity ports where appropriate. This typing is not contrary to the EIA standard and is quite appropriate for these logic devices.

First, the AND gate:

```
use work.BASICDEFS.all;
      --  the EIA basicdefs package

entity ANDm is
      generic(
            propagation_delay : time := 0 ns
            );
      port ( In1   : in logic_mv_vector ;
            out1   : out logic_mv := 'U'
            );
end ANDm;
```

As for all components, a behavioral architecture was developed purely to permit VHDL simulation. The AND gate's archtecture is:

```
architecture BEHAVIORAL of ANDm is
begin

      out1 <= and_bw( In1 ) after propagation_delay;

end BEHAVIORAL;
```

The function and_bw() is defined in the EIA's BASICDEFS package.

The second example is a clocked D flip-flop. Again, its architectural body was developed to support VHDL simulation. It is represented as:

```
use work.BASICDEFS.all;
      --  the EIA basicdefs package

entity D_ff is
   generic ( propagation_delay        : time    := 0 ns );
   port (
          D_in      : in logic_mv      := 'U';
          Q_out     : out logic_mv     := 'U';
          CLK       : in logic_mv      := 'U';
          Clear     : in logic_mv      := 'U';
          SET       : in logic_mv      := 'U'
          );
end D_ff;

architecture BEHAVIORAL of D_ff is

begin
   process (CLK)
   begin
      if (CLK'event and CLK = '1') then
         if ((Clear = '1') and (SET = '1')) then
            Q_out <= 'X' after propagation_delay; end if;
         if ((Clear = '0') and (SET = '1')) then
            Q_out <= '1' after propagation_delay ; end if;
         if ((Clear = '1') and (SET = '0')) then
            Q_out <= '0' after propagation_delay; end if;
         if ((Clear = '0') and (SET = '0')) then
            Q_out <= D_in after propagation_delay; end if;
         end if;
   end process;
end BEHAVIORAL;
```

The currently permitted components and their function are:

INVERTER	single input, single output inverter
AND2	two input AND gate.
ANDm	vectorized input AND gate
OR2	two input OR gate
ORm	vectorized input OR gate
NAND2	two input NAND gate
NANDm	vectorized input NAND gate
NOR2	two input NOR gate
NORm	vectorized input NOR gate
D_ff	Clocked D flip-flop with Set and Clear
D_ff_noclk	Asynchronous D flip-flop with SET and Clear

3.4.2 The Structural Architectural Body

The structural architectural body performs two roles. First, it completely specifies the sequential circuit in VHDL. This permits testing via the VHDL software environment. Second, by accomplishing the first goal, it contains the basic information required by the Senum, Pre_verif, and Verif software. This is accomplished via the component netlist, the initialization statements, and the D flip-flops. The initialization statements allow the UC Berkeley software access to the machine's initial state; the D flip-flops provide state information.

3.4.2.1 Instantiated Components

Instantiated components are declared in the VHDL norm:

```
name : component_name
            generic map ( generic_assignments );
            port map ( port_assignments );
```

Currently, only positional association of the signal to port assignments is permited. A sample component is:

```
inv1 : inverter
            port map (Q2, Q2not);
```

3.4.2.2 Bus Support

Busses within a design are required when vectorized-input logic gates are used. They are declared as logic_mv_vector type in the architecture's declarative block and comprised of the concatenation of scalar signals within the architecture. An example is:

```
BUS <= Instruction & Q1 & Q2not;
```

The bus is then used as an input to the vectorized logic gates:

```
gate1: ANDm
       port map ( BUS, gate1_out );
```

3.4.2.3 Initialization

In order to properly simulate via VHDL or verify via the UC Berkeley software, the sequential circuit's initial or starting state must be known. This can be accomplished several ways in VHDL. Multiplexed logic driving the flip-flops, SET and CLEAR lines on the flip-flops, etc -- but each clearly specifies the hardware and its interconnections. The chosen approach provides the initial state information while not specifying the initialization method. Initialization is accomplished via signal assignment statements within the architecture. Two methods are permited. Each method assigns "initial values" to the flip-flop outputs (designated as q's); this initial value is removed after the first clock triggers the flip-flop after some time_delay. The following examples set the initial state of the six flip-flop circuit to "101001". The first example explicity sets the intial state at simulation start; the second when the entity port signal "INITIALIZE" is a logical one:

Example One:

```
Qinit <= "101001", "ZZZZZZ" after time_delay;
q0 <= Qinit(0);
q1 <= Qinit(1);
q2 <= Qinit(2);
q3 <= Qinit(3);
q4 <= Qinit(4);
q5 <= Qinit(5);
```

Example Two:

```
with INITIALIZE select
      Qinit <= "101001" when '1',
            "ZZZZZZ" when others;
q0 <= Qinit(0);
q1 <= Qinit(1);
q2 <= Qinit(2);
q3 <= Qinit(3);
q4 <= Qinit(4);
q5 <= Qinit(5);
```

3.4.3 Structural Model Example

The following example is an implementation of a sequence detector using AND-OR logic. The circuit is intended to detect the input bit string "1001." Figure 3.7 shows the state machine diagram which models this circuit. From the diagram, the machine possesses four states which are implemented as two flip-flops. Further, state variable assigment is as follows:

State	Variable
Starting State	00
Detected 1	01
Detected 0	11
Detected Second 0	10

Figure 3.8 shows the schematic diagram of one implemetation of the sequence detector using D flip-flops and AND-OR gates.

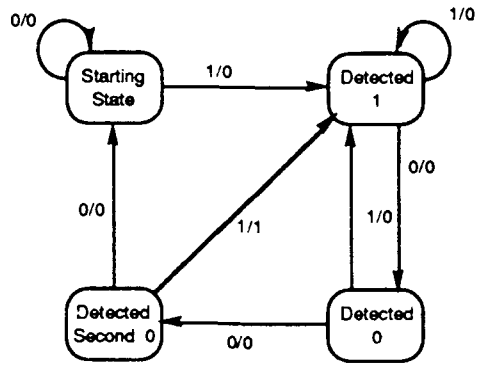


Figure 3.7. Sequence Detector State Diagram.

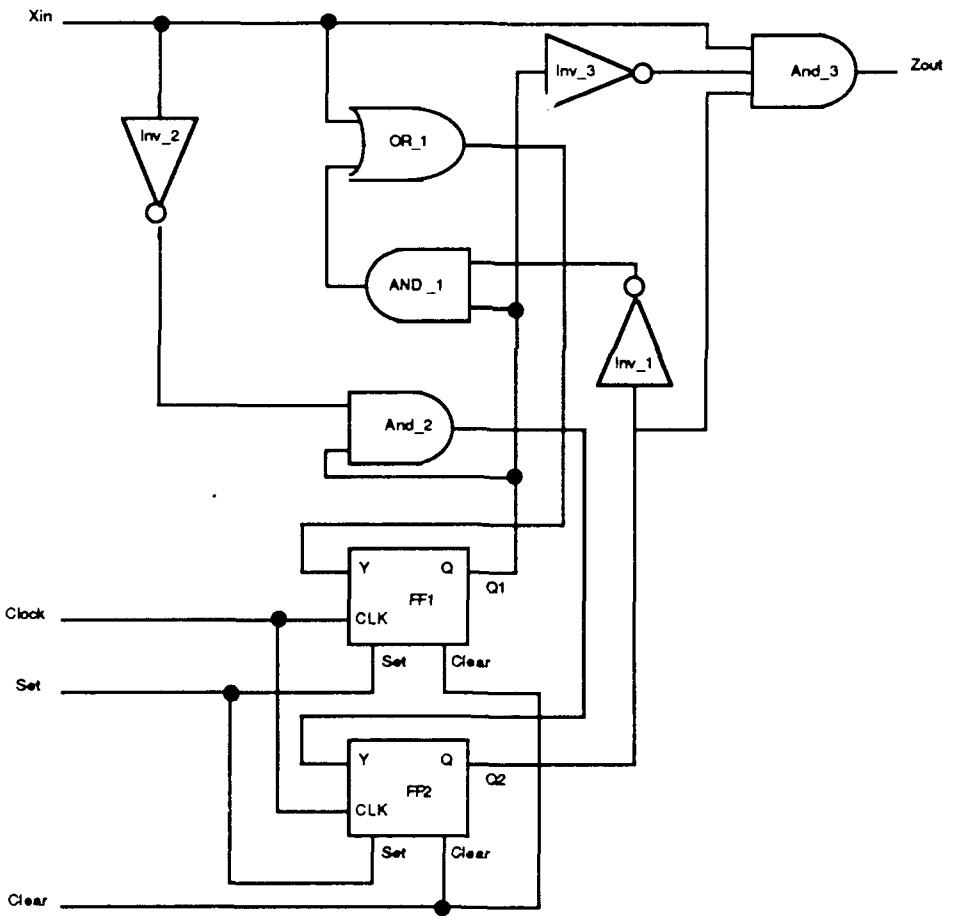


Figure 3.8. And-Or Implementation.

From Figure 3.8, the entity description is straighforward:

```
use WORK.basicdefs.all;
entity Sequence_Detector is
      -- generic ( );

      port (
            Xin     : in logic_mv    := 'U';
            CLOCK   : in logic_mv    := 'U';
            Zout    : out logic_mv   := 'U';
            SET     : in logic_mv    := 'U';
            CLEAR   : in logic_mv    := 'U'
           );
end Sequence_Detector;
```

Following the model guidelines, the architectural body sans declarative block follows. The complete VHDL Sequence Detector description along with test bench and sample simulation report are contained in Appendix C. Additionally, Appendix C contains a behavioral equivalent model using the proposed behavioral model of Section 3.3. Both the structural and behavioral designs were demonstrated equivalent via VHDL simulation on the same test bench. Both VHDL simulation results are included in Appendix C.

First, because of the vectored inputs to the AND and OR gates, signals internal to the structural description are declared within the architectural body's declarative block. These signals are:

```
signal  Y1, Y2, Q1, Q2 : logic_mv := 'U';

signal  Xnot, Q1not, Q2not,
        AND1_output : logic_mv := 'U';

signal  AND1_input, AND2_input,
        OR1_input : logic_mv_vector (1 downto 0) := "UU";

signal  AND3_input : logic_mv_vector (2 downto 0)
                                    := "UUU";


signal  Qinit : Wired_Outputs logic_mv_vector (1 downto 0)
                                    := "UU";
```

Next, the initialization signal is constructed as:

```
Qinit <= "101001", "ZZZZZZ" after 10ns;
q0 <= Qinit(0);
q1 <= Qinit(1);
q2 <= Qinit(2);
q3 <= Qinit(3);
q4 <= Qinit(4);
q5 <= Qinit(5);
```

Finally, the following VHDL code reflects the schematic diagram of Figure 3.8.

```
inv1 : inverter
          port map (Q2, Q2not);

inv2 : inverter
          port map (Xin, Xnot);

inv3 : inverter
          port map (Q1, Q1not);

---  Logic to derive Y1 and Y0:

    AND1_input <= Q1 & Q2not;

    AND1 : ANDm
          port map ( AND1_input, AND1_output );

    OR1_input <= Xin & AND1_output;

    OR1 : ORm
          port map ( OR1_input, Y1 );

    AND2_input <= Xnot & Q1;

    AND2 : ANDm
           port map ( AND2_input, Y2 );

--- LOGIC to derive Zout

    AND3_input <= Xin & Q1not & Q2;

    AND3 : ANDm
           port map ( AND3_input, Zout );

--- Registers

FF1 : D_ff
          port map ( Y1, Q1, CLOCK, CLEAR, SET );

FF2 : D_ff
          port map ( Y2, Q2, CLOCK, CLEAR, SET  );
```

The entire VHDL code for this example can be found in appendix C.

4 Verification Software

This chapter not only describes the modifications made to the existing UC Berkeley verification software in order for this software to utilize VHDL, but also, details the translator created for this thesis in order to translate behavioral to structural VHDL. The software modifications made enable the pre_verif software to accept structural VHDL designs expressed in the structural format of Chapter 3; the translator transforms sequential circuit specifications expressed in the behavioral VHDL model of Chapter 3 into an equivalent structural VHDL design. This new structural description can then be verified against another structural description. First the verification software environment is presented overviewing the relationship of the software components to the verification process. Then, the pre_verif modifications are presented followed by the behavioral to structural (b2s) translator software.

4.1 The Verification Software Environment

This section describes the verification software environment consisting of UC Berkeley's pre_verif and verif software tools and the b2s software. An example verification process of two sequential circuits, one described using the behavioral VHDL model and the other using the structural VHDL model, is used to describe the verification software environment. Figure 4.1 represents the functional relationships of the software components of the verification environment. The verification proceeds as follows. As depicted in Figure 4.1, the behavioral model is contained in file 1; the structural model in file 2. First, the behavioral model is translated into a structural model via the b2s software. Next, pre_verif generates a verif input file from the new structural model of the behavioral description. Pre_verif is executed again, but now a second verif input file is generated from the second structural model contained in file 2. Finally, the verif software is executed on the two verif input files to show the equivalence or non-equivalence of

the two sequential circuits. Refer to Appendix D for a more in-depth step through of the verification process.

4.2 Structural Translation

The structural VHDL translation by pre_verif is precipitated by the notion that a structural VHDL description contains the requisite information contained in a similar design expressed in the original pre_verif UC Berkeley input netlist format. This section is divided in two parts. The first shows that the pre_verif input netlist format contains information equivalent to that of a similar VHDL description; the second provides both an explanation of the modifications made to pre_verif to accept a VHDL description and a detailed format for a VHDL description accepted by the new pre_verif.

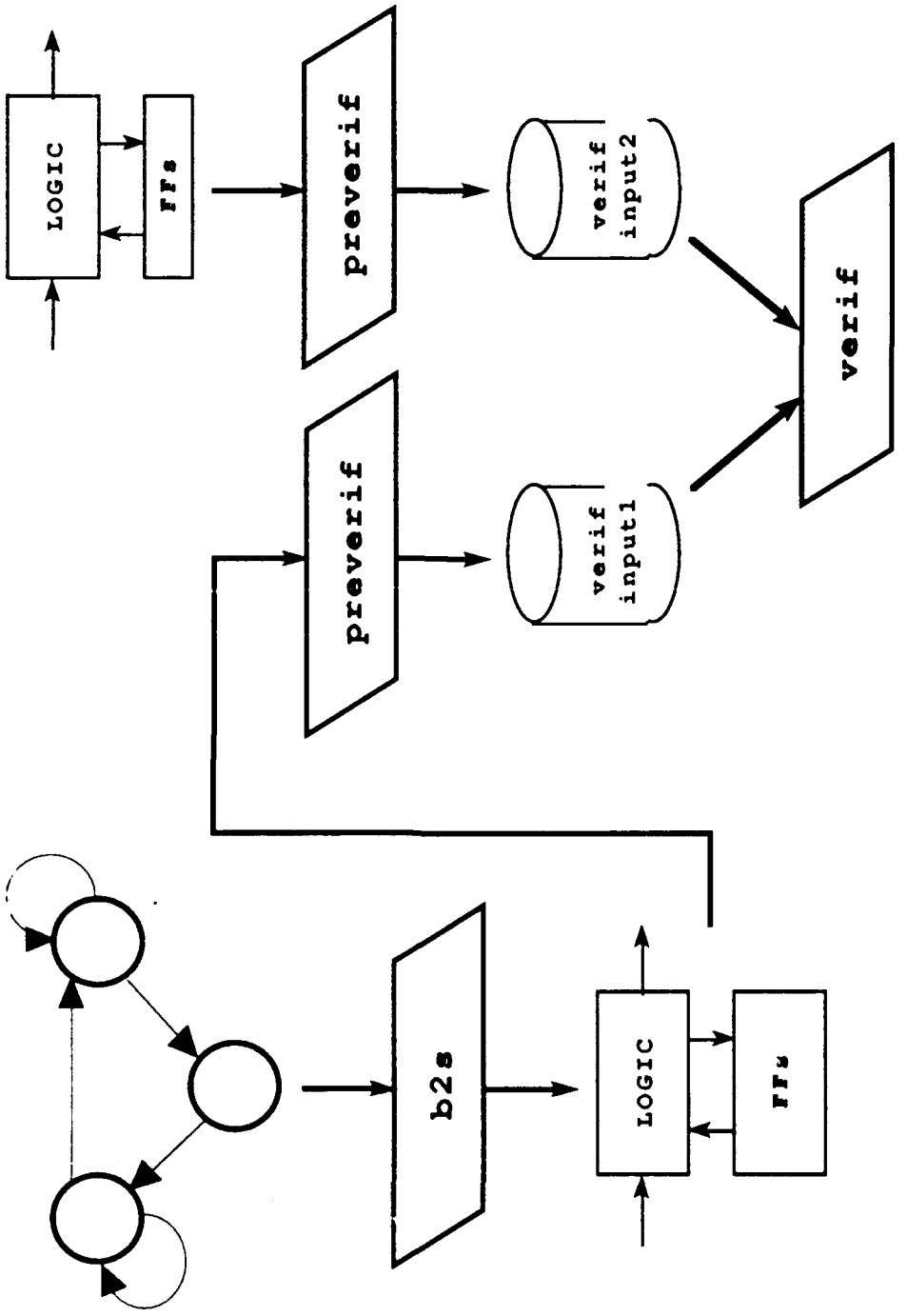


Figure 4.1 Verification process flow.

4.2.1 VHDL mappings to UC Berkeley Netlist

A pre_verif input file presents a structural description of a sequential circuit. The file contents can be divided into seven types of information which are:

comment lines (denoted by the symbol #),

combinational logic gate instantiations (denoted by the symbol g),

input signals (denoted by the symbol i),

output signals (denoted by the symbol o),

next state information (denoted by the symbol n),

present state information (denoted by the symbol p), and,

initialization information (denoted by the symbol l).

The input file is position sensitive regarding these symbols in that the first character on each new line *must* be with one of these seven symbols. Additionally, the first line of a pre_verif input file must contain the circuit name as:

name circuit_name

where circuit_name is the user defined name for the sequential circuit.

Inputs to the sequential circuit are defined as:

```
i circuit_input_1 circuit_input2 ... circuit_input_n
```

where circuit_input_1 through circuit_input_n are unique names defining the sequential circuit's input signals.

Circuit outputs are defined in a like manner:

```
o circuit_output_1 circuit_output2 ... circuit_output_n
```

where circuit_output_1 through circuit_output_n are unique names defining the sequential circuit's output signals.

A combinational logic element is defined as:

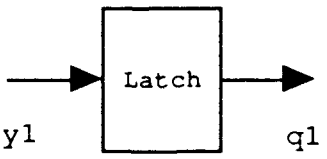
```
gname TYPE input_1 input_2 ... input_n ; output
```

where name is a unique user defined name to identify the particular logic element. TYPE is the element type: BUF (for buffer), AND, OR, NOT (for inverter), NOR, and NAND. The logic element may have any number of inputs (as signified by input1 input2 ... input_n) and only one output (as signified by output). Input and output lines must have unique names and are differentiated as input or output signals by a semi-colon. Those signal names before the semi-colon are inputs to the logic element; the signal name after is the output. Components may have any number of input signals but must have one, and only one, output.

A latch contains information concerning its present and next state. The latch's present state is its output line and its next state is its input line. A clock line is not included in the UC Berkeley format. The two UC Berkeley lines to describe a latch are:

```
ps q1
ns y1
```

These lines define the latch as:



Additionally, each latch's initial state must be set. This is accomplished by the I line with boolean logic 0s (zeros) or 1s (ones). By setting the latch's initial state, the overall initial state of the sequential circuit is also set.

The following short example utilizes each of pre_verif's various types. The example represents a sequential circuit that cycles through four states issuing a '1' output upon return to its initial state.

```
name Up_counter
#define inputs and outputs
i x1 x2
o z

#combinational logic
g1 not x1 ; x1not
g2 not x2 ; x2not
g3 not q1 ; q1not
g4 not q2 ; q2not

g5 and x1not x2 q1 ; and5
g6 and x1 x2not q1   ; and6
g7 and x1 x2not q2not ; and7
g8 or and5 and6 and7 ; y1

g9 and x1 q2 ; and9
g10 and x2 q2 ; and10
g11 and x1not x2 q1 ; and11
g12 or and9 and10 and11 ; y2

g13 and x1not x2not q1not q2 ; z

#first latch
ps q1
ns y1

#second latch
ps q2
ns y2

#initialization
I
00
```

Constructs within a VHDL structural design can be directly mapped to the input format detailed above. These mappings are presented first, followed by a complete VHDL structural model equivalent to the previous example sequential circuit expressed in UC Berkeley's format.

A VHDL entity description contains, at a minimum, the component's name and any input and output ports. The pre_verif format of:

```
name Up_counter
#define inputs and outputs
i x1 x2
o z
```

can be represented in VHDL utilizing EIA's basicdefs package to define port types as:

```
use work.BASICDEFS.all;
entity Up_counter is

-- define inputs and outputs
      port( x1 : in logic_mv;
            x2 : in logic_mv;
            z  : out logic_mv;
            q  : inout logic_mv
           );
end;
```

Further, component instantiations expressed as:

```
g4 not q2 ; q2not
```

can be represented in the VHDL structural model as:

```
g4 : inverter
      port map( q2, q2not );
```

In the case where a component has multiple inputs, the VHDL model allows signal concatenation permitting generic multiple input devices to be used. An example in the UC Berkeley format is:

```
g5 and x1not x2 q1 ; and5
```

Where x1not, x2, and q1 are the input signals to the AND gate g5. Using a generic-width input component as detailed in Chapter 3, this component instantiation can be expressed in VHDL as:

```
new_signal <= x1not & x2 & q1;
g5 : ANDm
      port map( new_signal, and5 );
```

Latches may be represented several ways (J-K, T, D, etc). For this effort all latches are represented as both synchronous or asynchronous D flip-flops. This clocking assumption is valid in that the UC Berkeley input format assumes the same; however, a clock signal line is not present

in the UC Berkeley format but is required in the equivalent VHDL description. The clock line not only fully specifies the synchronous sequential circuit, but also permits simulation using the VHDL software support environment. Further, two additional signals offered in the VHDL D flip-flop description are the SET and CLEAR lines. These lines allow a finer control of the VHDL description than that offered in the UC Berkeley format. A latch described first in pre_verif's and then in VHDL's format is:

```
ps q2
ns y2


FF1 : d_ff
      port map( y2, q2, CLOCK, SET, CLEAR );
```

Finally, circuit initialization expressed in UC Berekely format as:

```
I
00
```

can be expressed in VHDL in one of two ways:

```
Qinit <= "101001", "ZZZZZZ" after time_delay;
q0 <= Qinit(0);
q1 <= Qinit(1);
q2 <= Qinit(2);
q3 <= Qinit(3);
q4 <= Qinit(4);
q5 <= Qinit(5);
```

or

```
with INITIALIZE select
      Qinit <= "101001" when '1',
               "ZZZZZZ" when others;
q0 <= Qinit(0);
q1 <= Qinit(1);
q2 <= Qinit(2);
q3 <= Qinit(3);
q4 <= Qinit(4);
q5 <= Qinit(5);
```

In either case, the outputs of the flip-flops are set to an initial or reset value. The first example sets the flip-flops to their initial state upon the start of simulation; the second sets the flip-flops whenever the entity's INITIALIZE port is a logic '1' value.

A complete VHDL translation of the sequential circuit described earlier in this section utilizing UC Berkeley's format is then:

```
entity Up_counter is
      -- define inputs and outputs
      port( x1 : in logic_mv;
            x2 : in logic_mv;
            z  : out logic_mv;
            CLOCK : in logic_mv;
            INITIALIZE : in logic_mv
           );
end;

architecture STRUCTURAL of Up_counter is

 component inverter
      generic( propagation_delay : time := 0 ns );
      port (
            in1   : in logic_mv     := 'U';
            out1  : out logic_mv    := 'U'
            );
  end component;

      for all : inverter use
                  entity WORK.inverter(BEHAVIORAL);

  component ANDm
        generic( propagation_delay : time := 0 ns );
        port(In1  : in logic_vector_mv;
             out1 : out logic_mv := 'U'
             );
  end component;

      for all : ANDm use
                  entity WORK.ANDm(BEHAVIORAL);

  component ORm
        generic( propagation_delay : time := 0 ns );
        port(In1  : in logic_vector_mv;
             out1 : out logic_mv := 'U'
             );
  end component;

      for all : ORm use
                  entity WORK.ORm(BEHAVIORAL);
```

```
component AND2
        generic( propagation_delay : time := 0 ns );
        port(In1, In2    : in logic_mv := 'U';
             out1        : out logic_mv := 'U'
           );
  end component;

      for all : AND2 use
                  entity WORK.AND2(BEHAVIORAL);

  component OR2
        generic( propagation_delay : time := 0 ns );
        port(In1, In2    : in logic_mv := 'U';
             out1 : out logic_mv := 'U'
           );
  end component;

      for all : OR2 use
                  entity WORK.OR2(BEHAVIORAL);

  component D_ff
      generic ( propagation_delay   : time := 0 ns );
      port (
            D_in          : in logic_mv      := 'U';
            Q_out         : out logic_mv     := 'U';
            CLK           : in logic_mv      := 'U';
            Clear : in logic_mv      := 'U';
            SET           : in logic_mv      := 'U'
            );
  end component;

      for all : d_ff use
                  entity WORK.D_ff(BEHAVIORAL);

      signal new_signal1 : logic_mv = 'U';
      signal new_signal2 : logic_mv = 'U';
      signal new_signal3 : logic_mv = 'U';
      signal new_signal4 : logic_mv = 'U';
      signal new_signal5 : logic_mv = 'U';
      signal new_signal6 : logic_mv = 'U';
      signal new_signal7 : logic_mv = 'U';
      signal RESET, CLEAR : logic_mv = 'U';
      signal x1not, x2not, q1not, q2not : logic_mv = 'U';
      signal and5, and6, and7,
              and9, and10, and11 : logic_mv = 'U';

begin
      RESET <= '0';
      CLEAR <= '0';

      -- combinational logic
g1 : inverter
            port map( x1, x1not );
g2 : inverter
            port map( x2, x2not );
g3 : inverter
            port map( q1, q1not );
```

```
g4 : inverter
            port map( q2, q2not );

new_signal1 <= x1not & x2 & q1;
g5 : ANDm
            port map( new_signal1,  and5 );

new_signal2 <= x1 & x2not & q1;
g6 : ANDm
            port map( new_signal2, and6 );

new_signal3 <= x1 & x2not & q2not;
g7 : ANDm
            port map( new_signal3, and7 );

new_signal4 <= and5 & and6 & and7 ;
g8 : ORm
            port map( new_signal4, y1 );


g9 : AND2
            port map( x1, q2, and9 );

g10 : AND2
            port map( x2, q2, and10 );

new_signal5 <= x1not & x2 & q1;
g11 : ANDm
            port map( new_signal5, and11 );

new_signal6 <= and9 & and10 & and11;
g12 : or
            port map( new_signal6, y2 );

new_signal7 <= x1not & x2not & q1not & q2;
g13 : ANDm
            port map( new_signal7, z );

-- first latch
FF1 : d_ff
      port map( y1, q1, CLOCK, SET, CLEAR );

-- second latch
FF2 : d_ff
      port map( y2, q2, CLOCK, SET, CLEAR );

-- initialization
with INITIALIZE select
      Qinit <= "00" when '1',
               "ZZ" when others;
q0 <= Qinit(0);
q1 <= Qinit(1);

end STRUCTURAL;
```

This VHDL description is completely simulatable in the VHDL software support environment. As such, it contains more information than is required or specified by the UC Berkeley format. This additional information is acceptable, however, because modifications to pre_verif allow it to extract only that portion of the structural VHDL model required by the verification software. These modifications are discussed in the next section.

4.2.2 Pre_verif Modifications

The modifications made to pre_verif allow it to take in a VHDL structural description utilizing the relationships between VHDL and UC Berkeley's format as defined above. The VHDL input file required by pre_verif must contain both the entity and architectural body as depicted in the previous example. Pre_verif's output is directed to the file: verif.input.

All modifications made to the original pre_verif source are commented as such. One entirely new source file "vhdl_input.c" performs the VHDL translation. The new pre_verif software may be acquired by contacting AFIT's Electrical and Computer Engineering Department.

Pre_verif is invoked from the unix prompt by:

```
pre_verif [options] [infile] [> outfile]
```

Two options are required to process VHDL files. First, "-enum" must be included to perform cover enumeration of the input file; this option is required whether the input file is VHDL or UC Berkeley format. In order to process VHDL structural files, the option "-vhdl" must also be included. This was included such that if "-vhdl" is omitted, pre_verif will expect a UC Berkeley formatted input file rather than a VHDL file. The `[> outfile]` option redirects any output normally directed to the console screen into a file. A typical invocation of pre_verif is then:

```
pre_verif -enum -vhdl cpu.vhd
```

4.2.3 Pre_verif VHDL Constraints

Because of the proof of concept nature of the pre_verif modifications, several constraints have been imposed on the designer in utilizing the new VHDL option for pre_verif. Basically, these constraints reduce the free-form nature of the VHDL text file and are detailed as follows.

Within the entity description, all input and output signals must be the EIA BASICDEFS type of logic_mv. Additionally, only one signal is permitted per text line. The following VHDL entity description is correct.

```
entity Up_counter is
      -- define inputs and outputs
      port ( x1 : in logic_mv;
             x2 : in logic_mv;
             z  : out logic_mv;
             CLOCK : in logic_mv;
             INITIALIZE : in logic_mv
            );
end;
```

The following entities, although they represent correct VHDL, cannot be recognized by b2s. The first places the ");" symbol, which signifies the end of a port list, on the same line as a port declaration; the second declares the mode and type of multiple ports rather than one per line. These errors are boldfaced for clarity.

```
entity Up_counter is
      -- define inputs and outputs
      port( x1 : in logic_mv_vector( 5 downto 0);
             x2, x3 : in logic_mv;
             z  : out logic_mv;
             CLOCK : in logic_mv;
             INITIALIZE : in logic_mv);
end;
```

```
entity Up_counter is
      -- define inputs and outputs
      port( x1 : in logic_mv_vector( 5 downto 0);
             x2,
             x3 : in logic_mv;
             z  : out logic_mv;
             CLOCK : in logic_mv
            );
end;
```

Further, two signals, CLOCK and INITIALIZE, are reserved entity port names. CLOCK signifies the clocking signal used within the architecture for state-to-state transition and output synchronization; INITIALIZE names that signal used to reset or initialize the sequential circuit. CLOCK may be omitted; when missing an asynchronous sequential circuit is assumed. INITIALIZE, however, may NOT be omitted; this signal is required to establish the reset or initial state of the sequential circuit.

In the case of the architectural body, the VHDL modifications made by pre_verif ignore the entirety of the architecture's declarative block. Although it must be present to ensure a correct VHDL design, information contained within that portion of the design are not required by pre_verif or verif. Additional architectural body constraints are as follows.

Instantiated components must be in the following format:

```
g5 : ANDm
            port map( new_signal, and5 );
```

The following component instantiation forms are incorrect. The first places the port map on the same line as the component declaration; the second capitalizes the component's instantiated identifier. Further, the reserved VHDL "port map" must be lower case characters. The errors have been boldfaced for clarity.

```
g5 : ANDm port map( new_signal, and5 );

G5 : ANDm
     PORT MAP( new_signal, and5 );
```

Additionally, new signals declared within the architecture for use as inputs to the vectored input components (ANDm, ORm, etc) must be concatenated before use. The following example is correct:

```
new_signal <= q1 & input1 & q3not;

g5 : ANDm
      port map( new_signal, and5 );
```

The following, although equivalent in VHDL to the previous example, is an incorrect input format for pre_verif. All new signals which are comprised of concatenations of simple signals, must be created (by concatenation) before they are used in the components.

```
g5 : ANDm
      port map( new_signal, and5 );

new_signal <= q1 & input1 & q3not;
```

In general, when a problem with VHDL translation occurs using pre_verif, "white spaces" in the VHDL code should solve the problem. These white spaces may be blank spaces separating words or symbols, or, they may be carriage returns breaking a line up into smaller parts. One key enhancement to the *pre_verif* VHDL translator is the removal of all VHDL formatting constraints.

4.3 Behavioral Translation

This section details the behavioral to structural translator (hereafter referred to as "b2s") developed for this thesis. First, b2s's theory of operation is outlined followed by a description of the subset of the behavioral VHDL model which b2s can translate and, where applicable, the physical components which these represent.

4.3.1 The b2s Theory of Operation

As described in Chapter 2, one method of sequential circuit design is to capture the state machine representation as a state transition table and derive combinational logic and flip-flops from this state transition table. The b2s software operates in a similar manner. From a VHDL

behavioral model (currently limited to a Mealy machine as expressed in the behavioral model format of Chapter 3), b2s generates a state transition table (STT) representing the inputs, present state, next state, and outputs of the sequential circuit. From this table, a netlist of combinational logic and D flip-flops is created and formatted in the structural VHDL format of Chapter 3. All necessary architecturally internal signals and AND, OR, INVERTER, and D flip-flop components are declared within the architecture's declarative block. During the translation process, each and every minterm within the state transition table is generated; no algorithms are included for minimization of the combinational logic or optimization of the assigned binary state vectors. Additionally, certain constraints are placed on the "free form" VHDL style that b2s can translate; these constraints are enumerated in Section 4.3.3.

4.3.2 VHDL Behavioral to State Transition Table Mapping

This section describes the mapping of the VHDL behavioral model into the state transition table used by b2s to generate structural VHDL. The VHDL behavioral model is currently limited to a simple synchronous Mealy state machine; from the Mealy machine, a state transition table can be constructed which contains inputs, present state, next state, and output information.

From the entity description, b2s extracts input and output information. As in the pre_verif's VHDL structural input, CLOCK and INITIALIZE signals are reserved. The behavioral model's architectural body contains a transition process which provides the skeletal state transition graph of the machine, state blocks for each state of the machine, and additional optional blocks (or processes) which provide for initialization and clock synchronization. The state transition table is constructed from these portions of the architectural body in the following manner. First, from the VHDL transition process, the present state, next state, and transition columns of the state transition process are constructed. Each row of the state transition table represents one transition within the state machine. Therefore, for each transition condition encountered within the VHDL transition process, one row of the state transition table is

generated. The simple transition process of Section 3.3.1.1 is depicted in Figure 4.2 after the transition process has been processed. Next, input and output signals which are associated with each row of the state transition table are extracted from the individual state blocks which control each individual state's actions. Correspondence of the input and output signals to a particular row of the state transition table is derived by matching the transition signal assignment within the state block to the appropriate transition label of a row in the state transition table. This matching process is repeated for each input-transition pair within a state block. Figure 4.3 depicts b2s's addition of input and output information to a state transition table. This points out one key requirement levied on the behavioral VHDL description: within the architectural body, the transition process must precede the state blocks.

Inputs	Present State	Transition	Next State	Outputs
	First State	goto second State	Second State	
	First State	goto Third State	Third State	
	First State	others		
	Second State	goto First state	First State	
	Second State	goto Third state	First State	
	Second State	others		
	Third State	goto First State	First State	
	Third State	goto Second State	Second State	
	Third State	others		
	Unknown State	goto First State	First State	

Figure 4.2. State Transition Table after examining Transition Process.

Inputs	Present State	Transition	Next State	Outputs
0100	First State	goto second State	Second State	11
0010	First State	goto Third State	Third State	10
	First State	others		
0001	Second State	goto First state	First State	01
1000	Second State	goto Third state	First State	10
	Second State	others		
1111	Third State	goto First State	First State	01
1110	Third State	goto Second State	Second State	00
1100	Third State	others		
	Unknown State	goto First State	First State	

Figure 4.3. State Transition Table after Processing All State Blocks.

The translation performed by b2s from this state transition table into a state machine circuit comprised of combinational logic and flip-flops is straightforward. Although not presented here, the concepts of this process are readily available in (3) or (20).

4.3.3 VHDL Behavioral Constructs

The VHDL constructs which comprise the behavioral state machine model have been described in Chapter 3. This section presents the subset of these constructs currently recognized by b2s and details constraints imposed on their usage.

The current version of b2s translates a synchronous Mealy sequential circuit specified by a behavioral VHDL design into a structural version. As outlined in Section 4.2.2, this translation relies upon the transition process and the state blocks. The b2s software requires these VHDL constructs in a particular format within the architectural body. Additionally, an intialization block or process is required. Any clock process to permit synchronous state machine action is required for VHDL simulation, but is currently ignored by the b2s software.

The transition process must preceed all of the state blocks within the architecture. This process must take the form depicted in the following template (italicized words are user dependent; non-italicized words are expected to be in their depicted place as spelled and formatted):

```
process ( transition )
begin
      case Present_State is
            when First_State =>
                  case transition is
                        when transition2 =>
                              Next_State <= Second_State;
                        when transition3 =>
                              Next_State <= Third_State;
                        when others =>
                  end case;
            when Second_State =>
                  case transition is
                        when transition1 =>
                              Next_State <= First_State;
```

```
                        when transition3 =>
                              Next_State <= Third_State;
                        when others =>
                  end case;
            when Third_State =>
                  case transition is
                        when transition1 =>
                              Next_State <= First_State;
                        when transition2 =>
                              Next_State <= Second_State;
                        when others =>
                  end case;
            when others =>
      end case;
end process;
```

State blocks contain the input and output information for the particular state. They must take the form of the following template; again, italicized words are user defined. The signals input_1, input_2, and input_3 are the entity's input ports as defined by the designer; the signals out_1, out_2, and out_3 are the entity's output ports as defined by the designer.

```
FIRST_BLOCK: block ( Present_State = First_State )
      signal INPUTS : logic_mv_vector (2 downto 0);
begin
      inputs <=    input_1 & input_2 & input_3;

      process (GUARD, inputs)
      begin
      if ( GUARD ) then

          case inputs is
            when "000" =>
                  out_1 <= '1';
                  transition <= goto_Second_State;
            when "111" =>
                  transition <= goto_Third_State;
                  out_2 <= '1';
                  out_3 <= '1';
            when others =>
          end case;
      else
            transition <= Null;
            out_1 <= null;
            out_2 <= null;
            out_3 <= null;
      end if;
      end process;
end block FIRST_BLOCK;
```

4.4 Software Validation

Both the b2s software and the software modifications made to pre_verif were tested to ensure their correct operation. Most importantly, two key features were tested. First, the modified pre_verif's output file, generated from an input VHDL file, was checked in order to determine that it is equivalent to an output file generated by pre_verif from a UC Berkeley formatted file representing the same sequential circuit. Second, the structural VHDL design generated by b2s was verified equivalent to the behavioral VHDL specification through VHDL simulations. These tests were conducted on several different sequence detector designs. Each time the b2s and modified pre_verif software functioned correctly.

5 Model and Verification Examples

This chapter presents several examples of not only the behavioral and structural VHDL models proposed for sequential circuit modeling, but also presents several examples using the verification software to show the equivalence (or non-equivalence) of several sequential circuits expressed in VHDL. First, the behavioral model is demonstrated via the Test and Maintenance (JTM) bus developed as part of the Joint Integrated Avionics Working Group (JIAWG). Next, two sequential circuits using the structural model are presented including the verification of both. The first structural circuit is a sequence detector; the second is an eight-instruction CPU controller.

5.1 Behavioral Model

The following example exercises the capabilities of the behavioral model. Its purpose is to demonstrate various features of the behavioral model; as such, only portions of the examples can be simulated within the VHDL software support environment.

5.1.1 Test and Maintenance Bus

The test and maintenance bus (tm-bus) is part of a common avionics architecture which is to be used on the Air Force's Advanced Tactical Fighter (ATF), the Navy's A-12 Fleet Defense Fighter, and the Army's LHX Attack Helicopter program. The tm-bus provides a maintenance interface within the common avionics architecture (23).

5.1.2 Description

The tm-bus is a serial bus which transmits diagnostic control and status information between the modules interconnected via the tm-bus (23). Figure E.1 within Appendix E details the state transition graph representing the possible tm-bus states during information flow on the bus.

The tm-bus state diagram can be represented using the behavioral model's transition process. By providing the VHDL code as an adjunct to or a replacement of the English text specification, the tm-bus can be simulated by the contractor in order to ascertain the bus's operation. Further, portions of the VHDL code can be extracted for use in testing the tm-bus module's interface to the bus. This code is provided in Appendix E. As described in Chapter 3, this modeling method is superior to a standalone English text specification in that the English document can be open to interpretation; the VHDL cannot.

The behavioral VHDL sequential circuit model can also describe the modules which interface with the tm-bus. Within the tm-bus specification document, the functional description requires five pages: four for English text, one for a state diagram. As shown in Appendix E, the VHDL behavioral model can represent the same basic information in two pages.

Additional portions of a tm-bus module were described using the behavioral VHDL model. This modeling effort, performed for the tm-bus committee of the JIAWG, is described in Appendix E. Simulation testing was not attempted on these module portions as the details of the specification document which they represent were still being defined by the JIAWG tm-bus committee. Regardless, these behavioral code portions demonstrate the capabilities of the behavioral sequential circuit model in specifying a tm-bus module.

5.1.3 Skeletal Design

The behavioral VHDL model was used to create a skeletal specification of a tm-bus module. The skeletal specification consists of that information necessary to reconstruct the state

transition diagram of Appendix E; this includes the states and transitions within the state transition diagram. The specification is simulatable; but it does not implement the full functionality of a tm-bus module as detailed in the tm-bus specification document. A behavioral VHDL model specification of that detail was not only beyond the scope of this research effort but also prevented by ongoing revisions of the tm-bus specification document by the JIAWG tm-bus committee. Appendix E contains the skeletal specification of the tm-bus module. Additional portions of the tm-bus module were implemented using the behavioral VHDL sequential circuit model in order to demonstrate the model's capabilities; the tm-bus module contains multiple concurrent processes within each state along with state machines nested within states. Incorporating multiple concurrent processes and nested state machines were of prime interest to the tm-bus committee.

The behavioral model readily accommodates concurrent processes within states. Each state is represented as a VHDL block construct. Within its statement part, the block construct permits multiple VHDL process constructs (16). As an example, the tm-module's startup timer state requires three concurrent processes (23). The first process performs built-in module tests (named Sbit). A second process counts a specified number of clock pulses and then forces entry into the tm-bus module's first survivor state if Sbit is complete. Finally, a third process listens for information flow on the tm-bus. If an RMT command is received over the tm-bus, the third process interrupts the startup-timer and forces entry into the tm-bus module's slave state. Appendix E presents a VHDL specification of this state.

Finally, within one of the tm-bus module's state is another state machine. Although generalized here, a tm-bus specific nested state machine is presented in Appendix E. A state machine within another state can be represented by Figure 5.1.

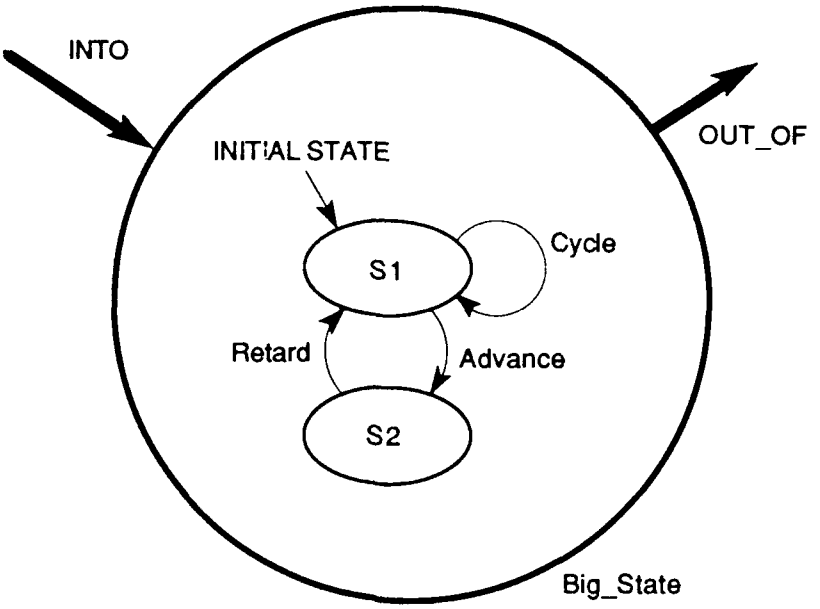


Figure 5.1. Nested State Machine.

A transition is made into "Big_State" when the transition condition INTO is assigned to the transition signal. When the transition occurs, the nested state machine is initialized to S1 and processing continues. A transition out of the "Big_State" is made when the transition condition "OUT_OF" is met and assigned to the transition signal. The behavioral VHDL model code to support this nested state implementation is as follows. The implementation assumes that the nested state's transition signal has its own resolution function and enumerated names of Retard, Cycle, and Advance.

```
Big_State_Block: block (Present_State = Big_State)
   signal  BS_Present_State : BS_States := Unknown_BS_State;
   signal  BS_Transition    : BS_Transition_Resolution
                                    BS_Transitions bus
                                    := no_BS_Transition;
   signal Leave_BS_State    : boolean := false;
begin

process
   if GUARD then  -- This process initializes the nested machine
                  -- and determines when to leave the Big_State.
      BS_Present_State <= S1;
      wait until (Leave_BS_State = TRUE);
```

```
      BS_Present_State <= Unknown_BS_State;
      Transition <= OUT_OF;
   else
      Transition <= null;
   end if;
end process;

process ( BS_transition ) begin
   case BS_Present_State is
      when S1 =>
         case BS_Transition is
            when Cycle =>   BS_Present_State <= S1;
            when Advance => BS_Present_State <= S2;
            when others =>
         end case;
      when S2 =>
         case BS_Transition is
            when Retard => BS_Present_State <= S1;
            when others =>
         end case;
      when others =>
   end case;
end process;

S1_State: block ( BS_Present_State = S1 )
begin
   process
      if GUARD then
         Do_some_action;
         BS_Transition <= Advance;
      else
         BS_Transition <= null;
      end if;
   end process;
end block S1_State;

BS_S2_State: block ( BS_Present_State = S2)
begin
   process
      if GUARD then
      Do_some_action;
         if Condition_1 then
                     BS_Transition <= goto_Xfer;
         else if Conditon_2 then
                     BS_Transition <= goto_Listen;
         else if Condition_3 then
                     Leave_BS_State <= TRUE;
         end if;
      else
         BS_Transition <= null;
      end if;
   end process;
end block BS_S2_State;


end block SLAVE_STATE;
```

In this code segment, the first process within the architectural block simply initializes the state machine and then waits until the Big_State's exit transition condition is met. Once met, it makes the assignment to the transition signal to fire the transition process exiting Big_State. The next process is the nested state machine's transition process. Its is identical in function to the transition process for a simple behavioral sequential circuit; it performs the state to state transitions within the nested state machine. The next two blocks represent the individual states of the nested state machine. As such, they perform the nested machine's actions and
as in the second block's case) set the exit flag, Leave_BS_State, signifying a transition must be made out of the Big_State.

5.2 Structural Model Examples

Two sequential circuits were implemented using the structural VHDL model of Chapter 3. The first sequential circuit implemented a sequence detector; the second implemented an eight-instruction CPU controller. The sequence detector was implemented three different ways; the CPU controller two. The intent of these examples was to not only demonstrate the abilities of the structural model, but also, to demonstrate the correct functionality of the verification software environment for both two equivalent and two non-equivalent sequential circuits. The following sections detail the results of these two structural modeling efforts.

5.2.1 Sequence Detector

The sequence detector structural VHDL examples were created to demonstrate the capabilities of the structural model and the ability of the verification software environment to show the equivalence of three separate sequence detector designs.

5.2.1.1 Description

The sequence detector was implemented as a single input, single output sequential circuit which detects the binary input string of "1001." The sequential circuit issues a '1' output whenever the string "1001" is detected. Figure 5.2 depicts the state transition graph for a Mealy machine implementing this function.

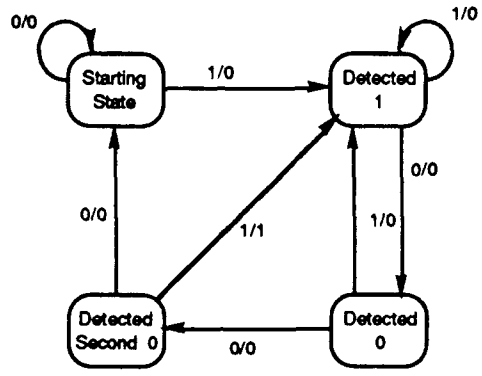


Figure 5.2. Sequence Detector.

5.2.1.2 Designs

The sequence detector was constructed using the structural VHDL model three different ways. First, using the state and state vector assignments of

State	Vector
Starting State	00
Detected 1	01
Detected 0	11
Detected Second 0	10

Two versions, one using AND-OR and another using NAND devices within the combinational logic, were constructed. Then, using an alternate state vector assignment of

State	Vector
Starting State	01
Detected 1	00
Detected 0	10
Detected Second 0	11

an alternate AND-OR version was constructed. Appendix F contains the structural VHDL code of these three versions.

5.2.2 Eight-Instruction CPU Controller

The eight-instruction CPU controller structural example was created to demonstrate the capabilities of the structural model and the ability of the verification software environment to *disprove* the equivalence of two non-equivalent circuit designs.

5.2.2.1 Description

Functionally, the eight instruction CPU controller is the same controller which was described using the behavioral VHDL model as detailed in Section 3.3.2.1. Unlike the behavioral implementation, the structural version contains sixteen states. The three additional states were required by each read or write to external memory.

5.2.2.2 Designs

The controller was implemented twice using the structural VHDL model -- one correct and one incorrect implementation. The incorrect implementation improperly decodes the JUMPZ instruction. For proper operation, the JUMPZ instruction checks the value of the accumulator. If the accumulator is zero, the JUMPZ instruction is performed; if the accumulator is not zero, the controller does not perform the JUMPZ instruction but fetches the next instruction for the CPU controller to decode. The error introduced in the second design forced the controller to always perform the JUMPZ instruction regardless of the accumulator's value. This error was introduced

by removing one transition from the controller's state transition graph. The bold arrow in Figure 5.3 points to the transition which was removed from the state transition graph. Appendix F contains the structural VHDL code for the correct CPU controller. Also, in Appendix F, is that portion of the improperly implemented controller's code which incorrectly decodes the JUMPZ instruction. This erroneous code replaces the JUMPZ decoding of the correct controller.

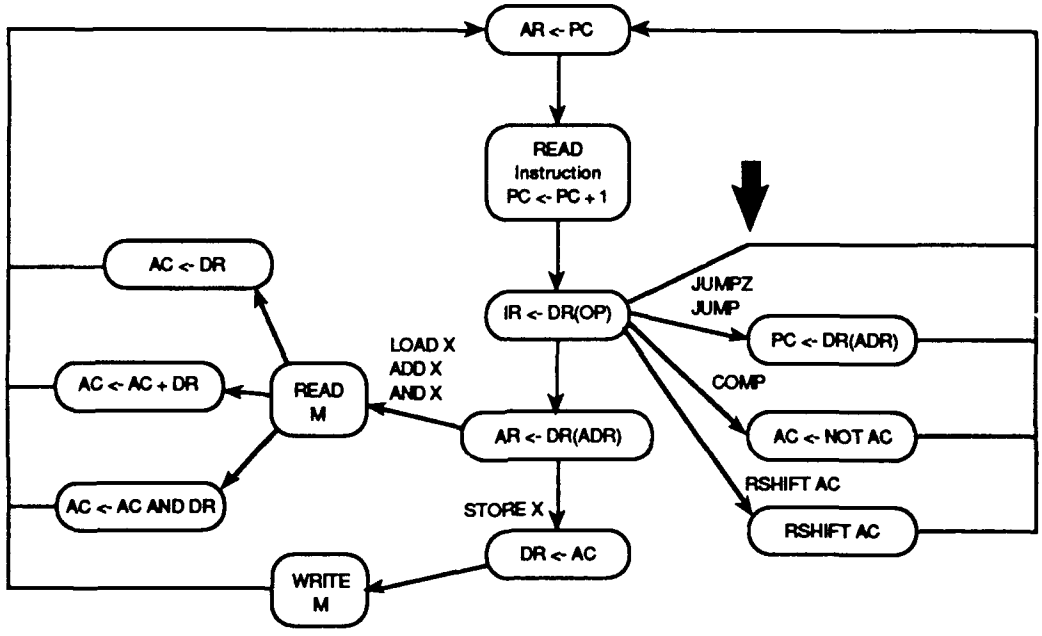


Figure 5.3. Eight Instruction CPU Controller Error Location.

5.3 Equivalence Verification

Equivalence verification was performed between the structural designs of Section 5.2. and between a behavioral and a structural sequence detector. This section details the results of these verification tests.

5.3.1 Structure to Structure Verification

The structure to structure verification was performed between the various sequence detector designs of Section 5.2.1 and between the two eight-instruction CPU controllers of Section 5.2.2. The results are presented here in that order.

5.3.1.1 Sequence Detectors

The equivalence of the three versions of the sequence detector was demonstrated in two ways. First, exhaustive simulation of the three sequential circuits was performed using the VHDL software support environment's simulator. The three sequence detectors were tested together on the same test bench. Appendix F contains the test bench and simulation report for one test of the sequence detectors. From these tests, the three sequence detectors were shown to be equivalent. Then, using the pre_verif and verif software, the three sequence detectors were again shown equivalent. As an additional test, this time intended to prove the veracity of the pre_verif VHDL translation routines, each sequence detector was duplicated using the UC Berkeley netlist format. Each VHDL version of the sequence detector was then verified against its equivalent UC Berkeley version. Each verification test ran successfully. See Appendix F for the UC Berkeley netlist versions of the sequence detector.

5.3.1.2 CPU Controllers

Like the sequence detector, the CPU controller was tested using both the VHDL software support environment's simulator and the verification software. Appendix F contains a sample simulation run which shows the correct operation of the correct CPU controller and the improper operation of the incorrect implementation. As in the sequence detector case, both CPU controllers were simulated on the same VHDL test bench.

The results of this example are quite promising in regards to the superiority of verification methods over exhaustive simulation. For comparison methods, both the VHDL simulation and

the verification processes began with the VHDL source code of the CPU controller. In simulation, both the correct and incorrect VHDL structural controllers were analyzed, model-generated, built, and simulated concurrently on the same test bench. A report file was generated which contained clocking, input, and output signal information. In verification, both the correct and incorrect VHDL structural controllers were translated via pre_verif and then verified via verif. Appendix F contains both the unix script files and results for the VHDL and verification runs. As can be seen from the runs, the VHDL simulation required approximately 30 minutes through report generation -- without any indication that the two controllers were not equivalent. The generated report file had yet to be scanned to determine equivalence. The verification process, on the other hand, terminated in approximately nine seconds and accurately reported that the two controllers were not equivalent. Additionally, once verif reported that the two controllers were not equivalent, it produced a set of input vectors which, when applied to each controller's state transition graph, correctly identify the incorrect state and state transition.

5.3.2 Behavior to Structure Verification

The behavior to structure verification was performed on the two-input, two-output, synchronous, sequential circuit of Figure 5.4. First, a behavioral description was created from the state transition diagram. Next, a structural description was created from Karnough maps of the circuit. Then, using the verification software, the two circuits were verified equivalent. Included in Appendix D is this verification example as an exercise. The VHDL behavioral and structural descriptions including the structural description created by the b2s software are included with Appendix D's example for completeness.

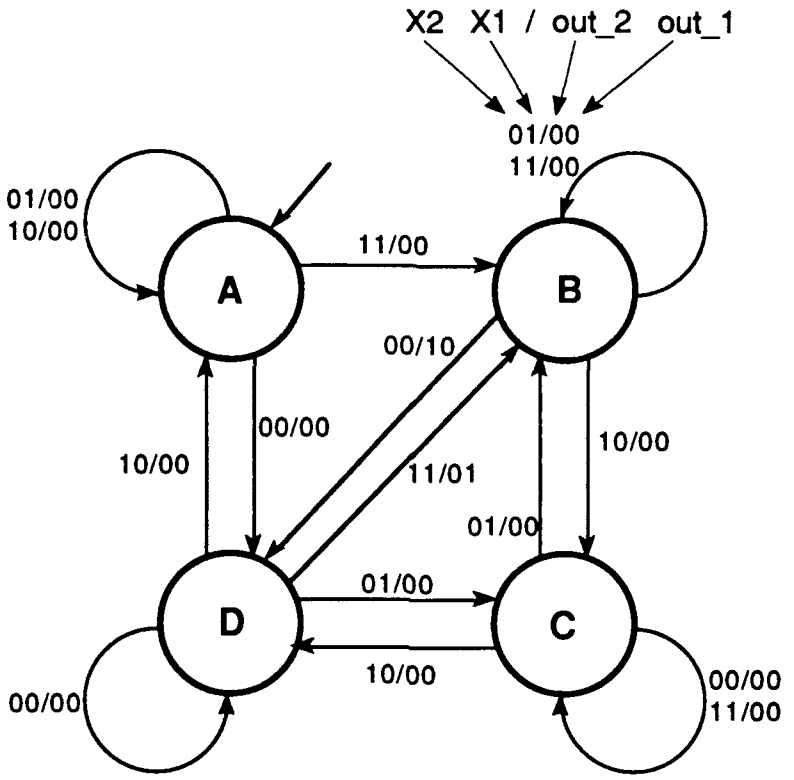


Figure 5.4. Two-input, Two-output, Synchronous Sequential Circuit.

The verification software verified that the two circuits were indeed equivalent. The software terminated with the results:

```
# Machine 1 inputs  2 outputs 2 latches 2
# Machine 2 inputs  2 outputs 2 latches 2
#Time to read in covers : 1.300000e-01 secs
#MACHINES ARE THE SAME
Number of states = 4
Number of edges  = 15
Number of entries  = 7
Number of save_difs  = 0
#Time for verification  : 7.000000e-02 secs
#Total user time        : 2.000000e-01 secs
```

6 Conclusions, Recommendations, and Summary

6.1 Conclusions

This section presents conclusions reached by the researcher concerning the sequential circuit models (behavioral and structural) and the integrated VHDL/verification process. First, the conclusions of several model examples are presented followed by the results of the verification process.

6.1.1 Models

As presented in Chapters 3 and 5, multiple sequential circuit examples were created using the behavioral and structural models. This section presents the results of both. The behavioral model is presented first followed by the structural.

6.1.1.1 The Behavioral Model

Muitiple sequential circuit examples were developed to prove the abilities of the behavioral VHDL model. Examples presented in this thesis included a synchronous Mealy, an asynchronous Moore (both in Appendices B and F), a hybrid sequential circuit incorporating multiple processes per state (Appendix B), and a hierarchical circuit containing nested state machines within states (Appendix E). In each case, the resulting circuit description exhibited specification advantages over previous efforts. Each simulated the correct behavior of the desired circuit and were more easily readable than designs expressed in the earlier specification styles presented in Chapter 2. In fact, the behavioral model was so well received, that it is currently being used by the DoD Joint Integrated Avionics Working Group's (JIAWG) test and maintenance bus (tm-bus) subcommittee to develop a VHDL specification of the tm-bus for the Air Force's and the Navy's ATF, and Army's LHX. When completed, this VHDL tm-bus

specification, which fully specifies and simulates the desired operation of the deliverable product, can replace the English-text portion of the contractual document to be delivered to the contractors.

6.1.1.2 The Structural Model

As presented in Chapters 3 and 5, the structural model was tested several ways. Three versions of a sequence detector and two versions of an eight-instruction CPU controller were developed (Appendices C and E). They were used not only to test the modeling ability of the structural model but also to exercise the verification software. From these, it was determined that the model is not unlike structural descriptions expressed in other design languages which contain a netlist of components and can be simulated in order to test the circuit's operation. From these examples, the structural model is shown adequate for designing sequential circuits using a limited set of standard components; but is hindered in expressiveness by this restricted set of components. As detailed in Chapter 3, these components were intentionally selected in order for the structural model to facilitate the proof of concept merger of the VHDL models to the UC Berkeley verification software.

6.1.2 Verification

The conclusions presented here are based on two structural-to-structural verifications and one behavioral-to-structural verification as discussed in Chapter 5. All three versions of a sequence detector were verified equivalent to each other via VHDL simulation and the verification software. Additionally, the verification software accurately identified the two eight-instruction CPU controllers as non-equivalent. For this latter case, the verification software also provided a set of input vectors which distinguished the different performance of the two controllers. Finally, the verification software accurately verified two sequence detectors; one described via the behavioral model and another via the structural model.

In both the structure-to-structure cases, the verification process accurately predicted the circuit's equivalence (or non-equivalence) in considerably less time than a VHDL simulation of the same components. As a case in point, the eight-instruction CPU controller required 29 minutes to perform VHDL analysis, model generation, and simulation; the verification software required less than 2 minutes. It's important to note that the 29 minutes required for VHDL simulation only produced a simulation report -- it did not include the time required to compare the simulation results in order to determine if the two controllers were equivalent or not. In less than 2 minutes, the verification process not only identified the two controllers as non-equivalent but also provided a set of input vectors, which, when applied to the CPU controllers during VHDL simulation, correctly delineated the different functionality of the two controllers.

For the behavior-to-structure case, the verification software was faster than the VHDL simulation again. More importantly, however, it demonstrates that behavioral circuit specifications can be merged into the verification process. This test aptly provided the proof of concept in the VHDL and verification methodology merger. Although this process currently accepts behavioral models which describe only synchronous Mealy sequential circuits -- its success demonstrates that structural design validation from a behavioral specification is possible. It warrants further research to exploit all the capabilities of VHDL and the verification software.

6.2 Recommendations

This section presents several recommendations for enhancements to or future research in the sequential circuit specification, design, and verification environment developed by this thesis. These recommendations fall into three categories: extending the VHDL language constructs permitted within both the behavioral and structural models, expanding the capabilities of the verification software, and, most importantly, incorporating timing parameters into both the behavioral and structural sequential circuit models and into the verification process. Each of these topics are discussed in the following sections.

6.2.1 VHDL Model Enhancements.

Enhancements should be made to both the behavioral and structural models proposed in this thesis in order to increase the models' fluency in circuit specification and design. As presented in Chapter 3, the models developed in this research currently use a limited set of the VHDL language constructs -- more research is necessary to incorporate additional language constructs in order to permit a more fluent specification and design capability yet maintain the models' clarity. Proposed enhancements and the research necessary to make these enhancements are presented first for the behavioral model and then for the structural.

6.2.1.1 Behavioral Model Enhancements.

For the behavioral model, research should focus on incorporating more VHDL constructs and timing information. Although it is quite easy to blatantly add more constructs to those permitted within the behavioral model, the research should determine which constructs not only correctly specify the behavior of the sequential circuit but also, in their usage, do not imply one particular hardware implementation. As an example, the following behaviorally-correct VHDL code portion, which counts two negative falling clock edges before taking some action, may imply that the designer use an up counter circuit when the actual circuit is implemented.

```
process ( Clock )
 variable clock_count : integer := 0;
begin
 if negedge( clock ) then
      clock_count := clock_count + 1;
      if clock_count = 2 then
       -- some action is performed such as:
       outputs <= "0001";
      end if;
 end if;
end process;
```

In place of this process, a simple VHDL wait statement, which does not imply counting up or down, may be superior in that it does not dictate a counter's particular implementation. A simple counter could then be:

```
wait for 2 * Clock_period;
```

Further research is required to specify starting, stopping, and programming counters specified as wait statements.

Timing information could be entered in several locations within the behavioral model -- future work should investigate if timing information should be entered at will or entered in specific locations within the behavioral model. As an example, the delay time which can be associated with state-to-state transitions can be specified in two places within the behavioral model. One location is inside the state block where the transition signal is assigned one of the enumerated values indicating a transition is to take place. Within the state block, a state-to-state transition delay time of 10 nanoseconds could be expressed as:

```
Transition <= goto_First_State after 10ns;
```

Alternatively, within a second location (the transition process where the next state is evaluated based upon the value of the `Transition` signal), the design could specify:

```
Next_State <= First_State after 10ns;
```

While either method results in the correct specification and simulation of a 10 nanosecond transition delay, research is warranted to determine if one method more clearly or succinctly specifies the sequential circuit. Further work here, and in other VHDL constructs, is required. The issue of incorporating this timing information into the verification process is discussed in Section 6.3.3.

Moving up a level of abstraction within the behavioral VHDL specification, the block construct has been shown appropriate for modeling individual states. Future research should investigate replacing the state block or processes within the state block with a VHDL concurrent

procedure call. For example, the following state, which currently is contained in its entirety in the behavioral model:

```
First_State : block ( Present_State = State_1 )
begin
   process ( GUARD , X1, X2 )
      if GUARD then
         -- Location of State activities if state is active.
      else
         -- etc...
      end if;
   end process;
end block First_State;
```

would be replaced by:

```
First_State: First_State_Procedure(Present_State, X1, X2, Out_1);
```

where `Present_state`, `X2`, `X2`, and `Out_1` would be signals passed in and out of the procedure. The concurrent procedure call would be quite appropriate in large designs in that the procedures' contents would be located in separate VHDL packages which the architectural body references via the VHDL use construct. Additional research should not only investigate including concurrent procedure calls into the behavioral model but also, once incorporated, the research should extend the verification method to include the procedure calls. This would necessitate that the b2s software contain additional routines which search the designer's libraries and extract the appropriate concurrent procedure body information in order to generate the state transition table.

Finally, from a global system design perspective, the behavioral VHDL model is intended solely for specifying sequential circuits. There is no reason why a more omniscient behavioral model, such as that depicted in Figure 6.1, could not be developed in which the sequential VHDL model would be a component. Although this would preclude the use of the verification

methods developed by this research, future research should be directed towards developing this overall system model and applying verification methods to it.

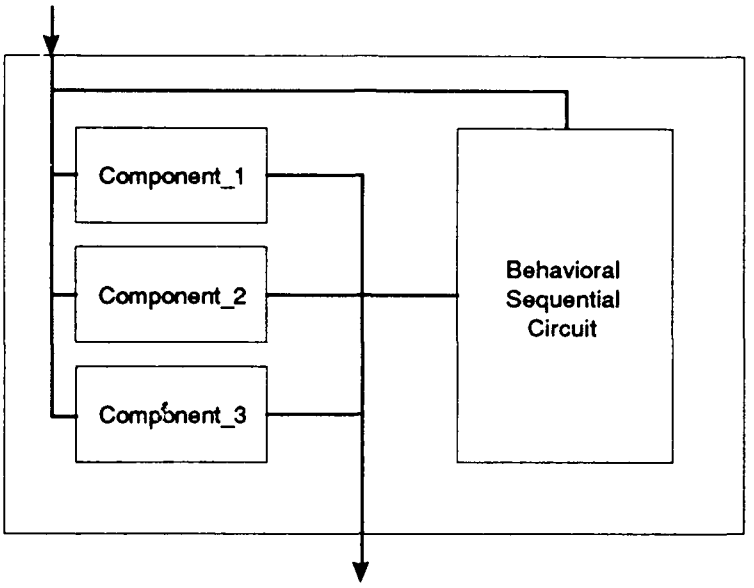


Figure 6.1. Behavioral VHDL Sequential Circuit within a System.

6.2.1.2 Structural Model Enhancements.

The structural model proposed by this thesis uses a small portion of the available language constructs in the VHDL; namely component instantiation, scalar and vector signals within the architectural body, and two initialization methods. Unlike the proposed behavioral enhancements which seek to increase the language constructs available to a designer, future enhancements to the structural model should focus on

(1) adding the "generate" language construct for instantiating arrays of regular components within the sequential circuit (ideal for multiple flip-flop instantiation),

(2) incorporating a larger component selection within the design (such as PALs, PLAs, ROMS, custom components, etc),

(3) refining methods for initializing the sequential circuit, and,

(4) including inter- and intra-component delay timing information.

Each of these additions would increase the structural model's fluency and capability in describing the sequential circuit as an interconnection of electronic components. Additionally, any enhancements to the structural model would require modifications to the pre_verif VHDL translator code in order to allow pre_verif to accept the enhanced structural VHDL design.

Finally, as in the behavioral model's case, there is no reason why a more omniscient structural model, similar in concept to the behavioral one depicted in Figure 6.1, could not be developed in which the structural sequential VHDL model would be a component. In Figure 6.1, the structural sequential VHDL model would replace the behavioral model. Although this would preclude the use of the verification methods developed by this research, future research should be directed towards developing this overall system model and applying verification methods to it.

6.2.2 Verification Software Enhancements

Enhancements should be made to both the pre_verif and b2s software tools. Enhancements to pre_verif should center on its ability to translate the structural VHDL model; enhancements to b2s should involve both its behavioral VHDL model translation capability and its output file formats. These modifications to pre_verif and b2s are discussed in the following sections. An additional enhancement common to both tools would be adding an ability to translate timing information which has been incorporated into the VHDL models; this modification is discussed in Section 6.3.3.

6.2.2.1 Enhancements to pre_verif

Enhancements to the pre_verif software should center on removing the current constraints on the format of the VHDL constructs which pre_verif recognizes. These constraints, as presented in Chapter 4, range from the simple lower-case character recognition of symbols such as "port map" or "entity" to enforced two line construct formatting of

```
g4 : ANDm
     port map ( input_signals, output_signal );
```

rather than

```
g4 : ANDm port map ( input_signals, output_signal );
```

Additionally, pre_verif should be extended in order to translate any enhancements incorporated into the structural VHDL model as proposed in Section 6.3.1.2. These modifications to pre_verif would greatly enhance the tools ease of use.

6.2.2.2 Extensions to b2s.

The b2s extensions proposed for future research involve three different areas. First, b2s should be extended to translate the additional VHDL constructs incorporated into the behavioral VHDL model as suggested in Section 6.3.1. Next, the structural VHDL synthesized by b2s should be optimized; this enhancement is discussed in Section 6.3.2.2.1. Finally, alternate b2s netlist output formats should be investigated; two are proposed in Section 6.3.2.2.2. This work to enhance b2s would greatly aid AFIT's VLSI design capabilities in allowing direct circuit synthesis from a behavioral VHDL description.

6.3.2.2.1 Structural VHDL Optimization.

In its current version, the b2s software produces combinational logic for the structural VHDL model which is not optimized from the behavioral circuit's VHDL specification. Each minterm within the sum of products equations, which are realized by the combinational logic, is explicitly enumerated. Further, b2s forms these minterms, whether or not they are required (ie, a don't care situation), from every signal within the state's input and present state vector(s). Finally, during the generation of the state transition table from the behavioral VHDL description, no state vector assignment optimization is performed. Future enhancements to b2s should incorporate some method to optimize the combinational logic.

The optimization could be accomplished in several ways. First, the minterms could be minimized. Minimization would reduce the number of variables required within minterms and, in turn, reduce the complexity of the combinational logic. Additionally, minterms which are common between any of the output(s) and/or next state(s) signals should be instantiated as one AND gate rather than the current method of instantiating an AND gate for each usage of the same minterm within the multiple sum of product equations. Finally, a method for optimizing state vector assignments should be investigated in order to minimize the combinational logic through judicious state vector assignment.

6.3.2.2.2 Alternate b2s Output Formats.

Alternative b2s output formats should be investigated in order to explore structural design synthesis from the behavioral VHDL specification. One alternative b2s output could be a SPICE-based structural netlist. As mentioned above, another output format could be a MAGIC, or CIF based, macro-cell chip layout synthesized directly from the behavioral VHDL model. Each alternative output format would provide a logical "next step" in the design process originating from the behavioral VHDL model.

The SPICE-based netlist would provide the capability of performing an in depth analog-based timing, fanout, and power consumption analysis of the sequential circuit. This analysis is currently not available in VHDL. Results from the SPICE simulations could then be back annotated into the behavioral or structural VHDL sequential circuit descriptions in order to produce a quite complete VHDL specification or design of the desired sequential circuit.

The MAGIC or CIF based macro-cell netlist would provide the capability of synthesizing the sequential circuit on a MOSIS chip directly from the VHDL behavioral description. Automating this synthesis step would eliminate any MAGIC design errors which are inadvertently introduced during the manual layout step currently performed at AFIT. Additionally, automating this process

would eliminate the time consuming manual layout step and permit more time for the design and testing of the sequential circuit's behavior.

The generation of either alternate output format is not a simple matter. In the case of the SPICE-based output, research would be necessary in order to properly characterize the SPICE structures of the current VHDL structural components (AND, OR, NOR, NAND, flip-flops, etc). Although the structural components could be represented in SPICE as subcircuits, base line performances and sizes including a mechanism for dealing with 2, 3, 4, or more input signals should be developed. Further, some methodology must be derived in order to back annotate information gleaned from the analog simulations into the behavioral (or structural) VHDL descriptions. Key to the back annotation would be deriving a method to incorporate inter-component delays introduced by resistive and capacitive affects of wire routing on the VLSI layout.

For the MAGIC or CIF based representations, automated cell placement and routing may be an intractable issue. Research is required in these areas. Additionally, this representation shares with the SPICE format the need to represent the structural components as macro-cells. Also, as with the SPICE version, AFIT lacks a macro-cell library of MAGIC components; each chip layout is a fully custom, hand-crafted design.

6.2.3 Incorporating Timing Into the Verification Process.

Currently, the verification process used in this thesis does not incorporate any notion of timing delays brought about by signal propagation delays internal to or propagation delays between devices within the sequential circuit. VHDL possesses language constructs for introducing timing delays into both the behavioral or structural VHDL models; their inclusion in the behavioral and structural models is discussed in Section 6.3.1.1 and Section 6.3.1.2. Using these constructs, both the behavioral and structural VHDL sequential circuit models can accurately portray the timing characteristics of an actual circuit. Given their inclusion within the

VHDL models, this section presents one possible method for verifying not only the functional but also the timing equivalence of two sequential circuits which incorporate the timing delay constructs of VHDL.

First, a concise definition of "timing equivalence" should be derived. Two circuits may be logically equivalent producing the same output(s) for any given input(s) and present state condition, but the output(s) may be valid at vastly different times. The first circuit may produce a valid output after 10 nanoseconds while the second may produce the same output after 2 days. Functionally, these two outputs are equivalent; but the two circuits are hardly interchangeably equivalent. When the sequential circuit is synchronous, the "timing equivalence" definition may involve the output delay times when measured from the appropriate edge (or level) of the synchronizing clock. The asynchronous circuit may involve a combination of the timing delays for the output(s) and any time delays required to transition from state to state. Finally, the "timing equivalence" definition may affect the manner in which a check for timing equivalence is incorporated and/or performed in the verification software.

One possible approach is presented as follows. Figure 6.2 shows a representative state transition table for a Mealy machine which incorporates propagation delay time. In the figure, T1 represents the transition delay time required to transition from the present state into the next state as measured from the time when the inputs and present state are valid. T2 represents delay time required before the output is valid as measured from the time when the inputs and present state necessary to determine the output are present. This state transition table can be readily constructed from either the behavioral or structural VHDL models of a sequential circuit.

State Transition Table:					
Inputs	Present State	Next State	T1	Outputs	T2
0 -	000	1 - 1	1ns	110	3ns
1 -	1 - 1	0 1 -	2ns	100	2ns
0 -	0 1 -	100	1ns	000	4ns
- -	100	100	4ns	110	5ns
i -	100	100	5ns	000	2ns
1 -	110	100	1ns	100	1ns
00	100	100	3ns	001	2ns
- 1	100	111	1ns	111	2ns
00	1 - 1	0 1 -	1ns	100	2ns
1 -	000	100	1ns	001	2ns

Figure 6.2. Sample State Transition Table Incorporating Timing Information.

The manner in which the timing information represented by T1 and T2 is incorporated into the two VHDL models should receive close attention. For the behavioral model, these delay times may be introduced in several portions of the design, as represented in Figure 6.3. Whether the model portrays a synchronous or asynchronous circuit may further determine the appropriate method of inserting this timing information into the behavioral model. Additionally, for the structural model, a mechanism for determining the lumped sum propagation delay time and wiring delay time between components, as depicted in Figure 6.4, must be derived. The VHDL structural model does not include the architectural bodies of the components which are instantiated within the structural model. This would necessitate that the pre_verif software contain additional routines which search the designer's component libraries and extract the appropriate component timing information.

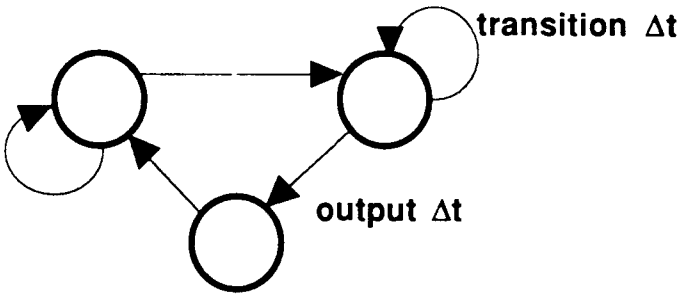


Figure 6.3. Delay Time Insertion into the Behavioral Model.

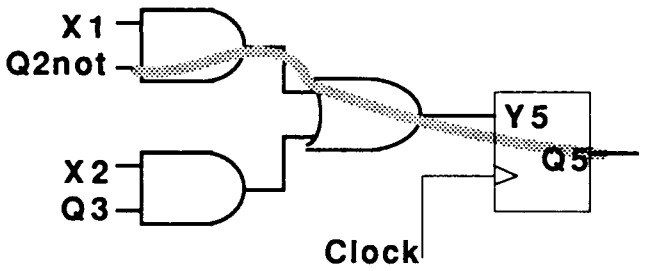


Figure 6.4. Lumped Sum Propagation Delay.

For the purposes of this example, "timing equivalence" will be defined such that each signal (both output and next state) must be identical between the two circuits under consideration. In reality, this may be too stringent a requirement; plus and minus time tolerance ranges about some desired valid output time may be more appropriate. Figure 6.5 depicts this tolerance range as a greyed region about two signals of interest.

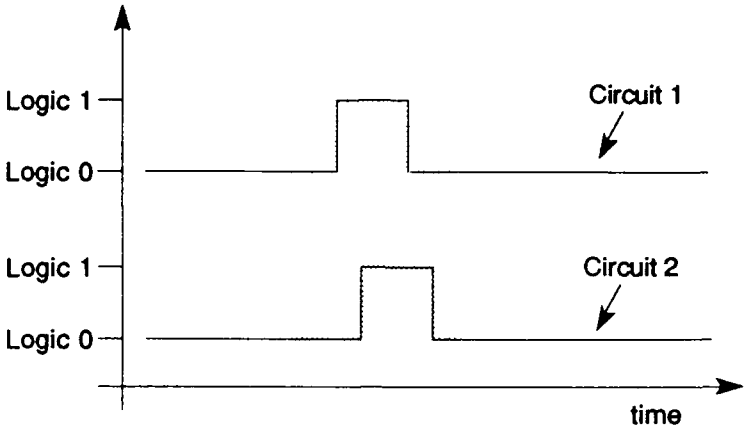


Figure 6.5. Equivalent Output Signals within a Tolerance Region.

One possible solution for the verification software may proceed as follows. As the verification software steps through each state pair of the two sequential circuits searching for a differentiating sequence to determine their equivalence or non-equivalence (as detailed in Chapter 2), the delay times for the particular output or state transition of the two machines can be compared. If these times are equivalent or within some acceptable time tolerance, the machines are considered identical and verification can proceed for the next state pair. If these times are different, the machines are different and verification may terminate.

6.3 Summary

This thesis has presented a solution to the problem of specifying, designing, and verifying sequential circuits. The solution is a merger of the specification and design capabilities of the VHDL with a known verification method in order to solve the design and verification problem of sequential circuits. Both goals of the research were met. The first goal was to determine appropriate VHDL language constructs for behavioral and structural modeling of a sequential circuit. This goal produced two VHDL-based models: one for behavioral specification and a another for structural design. The second goal was to apply the verification techniques of UC Berkeley's verif software to sequential circuits portrayed via the behavioral and structural VHDL

models. The product is a set of software tools, b2s, pre_verif, and verif, which compare two sequential circuits described via the VHDL models. These models and software tools have been tested and have been shown to be quite appropriate for solving the specification, design, and verification problem of sequential circuits. Finally, recommendations have been offered for improving both the VHDL models and the verification software.

APPENDIX A. EIA COMMERCIAL COMPONENT MODEL SPECIFICATION SP-2229

This appendix contains the seven level logic definitions and ancillary support functions as proposed for standardization by the Electronic Industry Association. For further information regarding this proposed standard and its use, consult the EIA specification document, SP-2229.

```
package BASICDEFS is
-------------------------------------------------------------------------
    The following is a preliminary definition of the basic logic system
--  and associated operators/functions used for the EIA VHDL Model
--  Commercial Component Specification.
--
--  Created by Dave Cantwell/Hughes       (714)-670-4677
--         & Len Finegold/General Dynamics
--
--  COPYRIGHT C Hughes Aircraft Co. 1989
--  Created 1/25/90
--  Version 0.1
-------------------------------------------------------------------------
   Type logic_mv is ('U', 'X', '0', '1', 'Z', 'L', 'H');
   Type logic_vector_mv is array (natural range <>) of logic_mv;
   Type logic_mv_table is array (logic_mv, logic_mv) of logic_mv;
---
--    Logic conversion functions
---
-------------------------------------------------------------------------
      SCALAR FUNCTIONS
-------------------------------------------------------------------------
      OVERLOADED OPERATORS
FUNCTION "and"    ( i1, i2 : logic_mv )    RETURN logic_mv ;
FUNCTION "nand"   ( i1, i2 : logic_mv )    RETURN logic_mv ;
FUNCTION "or"     ( i1, i2 : logic_mv )    RETURN logic_mv ;
FUNCTION "nor"    ( i1, i2 : logic_mv )    RETURN logic_mv ;
FUNCTION "xor"    ( i1, i2 : logic_mv )    RETURN logic_mv ;
FUNCTION "not"    ( i1     : logic_mv )    RETURN logic_mv ;
-- NOT A PREDEFINED OPERATOR, THUS IS NOT OVERLOADED.
FUNCTION xnor     ( i1, i2 : logic_mv )    RETURN logic_mv ;
-------------------------------------------------------------------------
      VECTORIZED FUNCTIONS
-------------------------------------------------------------------------
FUNCTION "and"    ( i1, i2 : logic_vector_mv )  RETURN logic_vector_mv ;
FUNCTION "nand"   ( i1, i2 : logic_vector_mv )  RETURN logic_vector_mv ;
FUNCTION "or"     ( i1, i2 : logic_vector_mv )  RETURN logic_vector_mv ;
FUNCTION "nor"    ( i1, i2 : logic_vector_mv )  RETURN logic_vector_mv ;
FUNCTION "xor"    ( i1, i2 : logic_vector_mv )  RETURN logic_vector_mv ;
FUNCTION "not"    ( i1     : logic_vector_mv )  RETURN logic_vector_mv ;
-- NOT A PREDEFINED OPERATOR, THUS IS NOT OVERLOADED.
FUNCTION xnor     ( i1, i2 : logic_vector_mv )  RETURN logic_vector_mv ;
--
--  BIT-WISE REDUCTION FUNCTIONS
--
FUNCTION and_bw   ( i1  : logic_vector_mv )     RETURN logic_mv;
FUNCTION nand_bw  ( i1  : logic_vector_mv )     RETURN logic_mv;
FUNCTION or_bw    ( i1  : logic_vector_mv )     RETURN logic_mv;
FUNCTION nor_bw   ( i1  : logic_vector_mv )     RETURN logic_mv;
FUNCTION xor_bw   ( i1  : logic_vector_mv )     RETURN logic_mv;
FUNCTION xnor_bw  ( i1  : logic_vector_mv )     RETURN logic_mv;
--
-------------------------------------------------------------------------
--  COMPARISON OPERATORS
-------------------------------------------------------------------------
FUNCTION "="    (i1, i2  : logic_mv)          RETURN logic_mv;
FUNCTION "/="   (i1, i2  : logic_mv)          RETURN logic_mv;
FUNCTION "="    (i1, i2  : logic_vector_mv)   RETURN logic_mv;
```

```
FUNCTION "/="  (i1, i2  : logic_vector_mv)   RETURN logic_mv;
-------------------------------------------------------------------------
BUS RESOLUTION FUNCTIONS
-------------------------------------------------------------------------
The following function shall be used for standard components
FUNCTION Wired_Outputs (signals : logic_vector_mv) RETURN logic_mv;
--  The following functions shall be used for on chip development ONLY
FUNCTION Wired_Or (Signals : logic_vector_mv) RETURN logic_mv;
FUNCTION Wired_And (signals : logic_vector_mv) RETURN logic_mv;
--
-------------------------------------------------------------------------
Miscellaneous Function(s)
-------------------------------------------------------------------------
   Function to translate H, L, or Z on inputs to 1, 0 or X respectively.
-- Example usage: in a RAM model, without this function, a HIGH or LOW
-- state would be stored internally from the bus and subsequently be
-- erroneaously driven onto the bus during a read operation.
FUNCTION Filter (input : logic_mv) RETURN logic_mv;
--
-------------------------------------------------------------------------
Signal transitions and relationships
-------------------------------------------------------------------------
FUNCTION Posedge ( signal s1 : logic_mv ) RETURN boolean;
FUNCTION negedge ( signal s1 : logic_mv ) RETURN boolean;
--  Posedge and NEgedge functions return TRUE on 0->1 or 1->0
--  transitions only
--
PROCEDURE Setup_check ( constant input_le : time;
                        constant time_spec : time;
                        constant message : string;
                        constant err_level : severity_level);

PROCEDURE Hold_check (  constant input_le : time;
                        constant time_spec : time;
                        constant message : string;
                        constant err_level : severity_level);

--
-- The following illustrates how to imbed setup and hold checks
-- procedures
-- into the VHDL models
--
--  DATA_CLOCK_SETUP: process
-- begin
-- wait on clk until do_timing_checks and posedge(clk,clk'last_value);
-- Setup_check (data'last_event, model_times.ts_data, "DATA to CLOCK",
--                        warning);
--  end process DATA_CLOCK_SETUP;
--
--  DATA_CLOCK_HOLD: process
--    begin
--    wait on clk until do_timing_checks andposedge(clk,clk'last_value);
--    Hold_check (data'last_event, model_times.th_data, "DATA to CLOCK",
--                        warning);
--  end process DATA_CLOCK_HOLD;
--
-------------------------------------------------------------------------
--    function name: F_delay
```

```
--    parameters:
--          in    newlv -- bit_mv   -- new logic value
--          in    delay01 --  time  -- 0->1 delay value
--          in    delay10 --  time  -- 1->0 delay value
--
--    returns:    The appropriate delay to be used, given the new value
--                and the 0-1 and 1-0 delays.
--
--   purpose:Compute the appropriate delay to be used for the transition
--                on an output port.
--
-------------------------------------------------------------------------
FUNCTION F_delay(   newlv    : IN logic_mv;
                    delay01  : IN time;
                    delay10  : in time     )  RETURN time;


end BASICDEFS;
```

```
package body BASICDEFS is
-------------------------------------------------------------------------
The following is a preliminary defintion of the basic logic system
--  and associated operators/functions used for the EIA VHDL Model
--  Commercial Component Specification.
--
--    Created by Dave Cantwell/Huges       (714-670-4677)
--                & Len Finegold/General Dynamics
--
--    COPYRIGHT C Hughes Aircraft Co. 1989
--    Created 1/25/90
--    Version 0.1
-------------------------------------------------------------------------
      CONSTANT DECLARATIONS FOR USE IN SIGNAL & VARIABLE ASSIGNMENTS.
-------------------------------------------------------------------------
constant MAX_SIZE          : POSITIVE := 32;     -- This is a deferred
                                                 -- constant which
                                                 -- should be initialized --
                                                 to the largest size
                                                 -- bus in design

constant UNINITIALIZED       : logic_mv    := 'U';
constant UNKNOWN             : logic_mv    := 'X';
constant ZERO                : logic_mv    := '0';
constant ONE                 : logic_mv    := '1';
constant HIGHZ               : logic_mv    := 'Z';
constant LOW                 : logic_mv    := 'L';
constant HIGH                : logic_mv    := 'H';
constant ALL_UNINITIALIZED   : logic_vector_mv (MAX_SIZE - 1 DOWNTO 0)
                                     := (others => UNINITIALIZED);
constant ALL_UNKNOWN         : logic_vector_mv ( MAX_SIZE - 1 DOWNTO 0)
                                     := (others => UNKNOWN);
constant ALL_ZERO            : logic_vector_mv ( MAX_SIZE - 1 downto 0)
                                     := (others => ZERO);
constant ALL_ONE             : logic_vector_mv ( MAX_SIZE - 1 downto 0)
                                     := (others => ONE);
constant ALL_HIGHZ           : logic_vector_mv ( MAX_SIZE - 1 downto 0)
                                     := (others => HIGHZ);
constant ALL_LOW             : logic_vector_mv ( MAX_SIZE - 1 downto 0)
                                     := (others => LOW);
constant ALL_HIGH            : logic_vector_mv ( MAX_SIZE - 1 downto 0)
                                     := (others => HIGH);
--
-------------------------------------------------------------------------
TYPE DECLARATIONS FOR USE IN SUBPROGRAMS BODIES
-------------------------------------------------------------------------
type logic_mv_array is array (logic_mv) of logic_mv;
--
-------------------------------------------------------------------------
SCALAR FUNCTIONS
-------------------------------------------------------------------------
FUNCTION "and" (i1, i2 : logic_mv ) RETURN logic_mv is

      constant TABLE : logic_mv_table :=
    (( UNKNOWN, UNKNOWN, ZERO,  UNKNOWN,  UNKNOWN,  ZERO,  UNKNOWN),
    (  UNKNOWN, UNKNOWN, ZERO,  UNKNOWN,  UNKNOWN,  ZERO,  UNKNOWN),
    (  ZERO,    ZERO,    ZERO,  ZERO,     ZERO,     ZERO,  ZERO),
    (  UNKNOWN, UNKNOWN, ZERO,  ONE,      UNKNOWN,  ZERO,  ONE),
```

```
    (  UNKNOWN, UNKNOWN, ZERO,  UNKNOWN,  UNKNOWN,  ZERO,  UNKNOWN),
    (  ZERO,    ZERO,    ZERO,  ZERO,     ZERO,     ZERO,  ZERO),
    (  UNKNOWN, UNKNOWN, ZERO,  ONE,      UNKNOWN,  ZERO,  ONE));
     begin

     RETURN Table( i1, i2);

     end "and";
-------------------------------------------------------------------------
FUNCTION "nand" ( i1, i2 : logic_mv ) RETURN logic_mv is

     constant TABLE : logic_mv_table :=
     (( UNKNOWN,  UNKNOWN, ONE,  UNKNOWN, UNKNOWN, ONE,  UNKNOWN),
        (UNKNOWN, UNKNOWN, ONE,  UNKNOWN, UNKNOWN, ONE,  UNKNOWN),
        (ONE,     ONE,     ONE,  ONE,     ONE,     ONE,  ONE),
        (UNKNOWN, UNKNOWN, ONE,  ZERO,    UNKNOWN, ONE,  ZERO),
        (UNKNOWN, UNKNOWN, ONE,  UNKNOWN, UNKNOWN, ONE,  UNKNOWN),
        (ONE,     ONE,     ONE,  ONE,     ONE,     ONE,  ONE),
        (UNKNOWN, UNKNOWN, ONE,  ZERO,    UNKNOWN, ONE,  ZERO));
     begin

     RETURN Table( i1, i2);

     end "nand";
-------------------------------------------------------------------------
FUNCTION "or" ( i1, i2 : logic_mv ) RETURN logic_mv is

     constant TABLE : logic_mv_table :=
     (( UNKNOWN,  UNKNOWN, UNKNOWN,  ONE,  UNKNOWN, UNKNOWN,  ONE),
        (UNKNOWN, UNKNOWN, UNKNOWN,  ONE,  UNKNOWN, UNKNOWN,  ONE),
        (UNKNOWN, UNKNOWN, ZERO,     ONE,  UNKNOWN, ZERO,     ONE),
        (ONE,     ONE,     ONE,      ONE,  ONE,     ONE,      ONE),
        (UNKNOWN, UNKNOWN, UNKNOWN,  ONE,  UNKNOWN, UNKNOWN,  ONE),
        (UNKNOWN, UNKNOWN, ZERO,     ONE,  UNKNOWN, ZERO,     ONE),
        (ONE,     ONE,     ONE,      ONE,  ONE,     ONE,      ONE));
     begin

     RETURN Table( i1, i2);

     end "or";

-------------------------------------------------------------------------

FUNCTION "nor" ( i1, i2 : logic_mv ) RETURN logic_mv is

    constant TABLE : logic_mv_table :=
    (( UNKNOWN,  UNKNOWN, UNKNOWN, ZERO, UNKNOWN,  UNKNOWN,  ZERO),
       (UNKNOWN, UNKNOWN, UNKNOWN, ZERO, UNKNOWN,  UNKNOWN,  ZERO),
       (UNKNOWN, UNKNOWN, ONE,     ZERO, UNKNOWN,  ONE,      ZERO),
       (ZERO,    ZERO,    ZERO,    ZERO, ZERO,     ZERO,     ZERO),
       (UNKNOWN, UNKNOWN, UNKNOWN, ZERO, UNKNOWN,  UNKNOWN,  ZERO),
       (UNKNOWN, UNKNOWN, ONE,     ZERO, UNKNOWN,  ONE,      ZERO),
       (ZERO,    ZERO,    ZERO,    ZERO, ZERO,     ZERO,     ZERO));
    begin

    RETURN Table( i1, i2);

    end "nor";
```

```
--------------------------------------------------------------------------
FUNCTION "xor" ( i1, i2 : logic_mv ) RETURN logic_mv is

     constant TABLE : logic_mv_table :=
     (( UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN),
     (UNKNOWN,   UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN),
     (UNKNOWN,   UNKNOWN, ZERO,    ONE,     UNKNOWN, ZERO,    ONE),
     (UNKNOWN,   UNKNOWN, ONE,     ZERO,    UNKNOWN, ONE,     ZERO),
     (UNKNOWN,   UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN),
     (UNKNOWN,   UNKNOWN, ZERO,    ONE,     UNKNOWN, ZERO,    ONE),
     (UNKNOWN,   UNKNOWN, ONE,     ZERO,    UNKNOWN, ONE,     ZERO));
     begin

     RETURN Table( i1, i2);

     end "xor";

--------------------------------------------------------------------------

FUNCTION xnor ( i1, i2 : logic_mv ) RETURN logic_mv is

   constant TABLE : logic_mv_table :=
   (( UNKNOWN,   UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN),
      (UNKNOWN,  UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN),
      (UNKNOWN,  UNKNOWN, ONE,     ZERO,    UNKNOWN, ONE,     ZERO),
      (UNKNOWN,  UNKNOWN, ZERO,    ONE,     UNKNOWN, ZERO,    ONE),
      (UNKNOWN,  UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN),
      (UNKNOWN,  UNKNOWN, ONE,     ZERO,    UNKNOWN, ONE,     ZERO),
      (UNKNOWN,  UNKNOWN, ZERO,    ONE,     UNKNOWN, ZERO,    ONE));
   begin

   RETURN Table( i1, i2);

   end xnor;

--------------------------------------------------------------------------
FUNCTION "not" ( i1 : logic_mv ) RETURN logic_mv is

   constant TABLE : logic_mv_array :=
   ( UNKNOWN,  UNKNOWN, ONE,  ZERO, UNKNOWN, ONE, ZERO);
   begin

   RETURN Table( i1);

   end "not";

FUNCTION "and" ( i1, i2 :  logic_vector_mv ) RETURN logic_vector_mv is
   alias Arg1 : logic_vector_mv ( 1 to i1'length ) is i1;
   alias Arg2 : logic_vector_mv ( 1 to i2'length ) is i2;
   variable Store : logic_Vector_mv ( 1 to i1'length );

   begin
      assert i1'length = i2'length report "Bus width Mismatch! "
                                    severity warning;
      for i in Store'range LOOP
         Store(i) := Arg1(i) and Arg2(i);
      end LOOP;
```

```
      return STORE;
   end "and";

-------------------------------------------------------------------------

FUNCTION "nand" ( i1, i2 : logic_vector_mv ) RETURN logic_vector_mv is
   alias Arg1 : logic_vector_mv ( 1 to i1'length ) is i1;
   alias Arg2 : logic_vector_mv ( 1 to i2'length ) is i2;
   variable Store : logic_Vector_mv ( 1 to i1'length );

   begin
      assert i1'length = i2'length report "Bus width Mismatch! "
                                          severity warning;
      for i in Store'range LOOP
         Store(i) := Arg1(i) nand Arg2(i);
      end LOOP;
      return STORE;
   end "nand";

-------------------------------------------------------------------------
FUNCTION "or" ( i1, i2 : logic_vector_mv ) RETURN logic_vector_mv is
   alias Arg1 : logic_vector_mv ( 1 to i1'length ) is i1;
   alias Arg2 : logic_vector_mv ( 1 to i2'length ) is i2;
   variable Store : logic_Vector_mv ( 1 to i1'length );

   begin
      assert i1'length = i2'length report "Bus width Mismatch! "
                                          severity warning;
      for i in Store'range LOOP
         Store(i) := Arg1(i) or Arg2(i);
      end LOOP;
      return STORE;
   end "or";

-------------------------------------------------------------------------
FUNCTION "nor" ( i1, i2 : logic_vector_mv ) RETURN logic_vector_mv is
   alias Arg1 : logic_vector_mv ( 1 to i1'length ) is i1;
   alias Arg2 : logic_vector_mv ( 1 to i2'length ) is i2;
   variable Store : logic_Vector_mv ( 1 to i1'length );

   begin
      assert i1'length = i2'length report "Bus width Mismatch! "
                                          severity warning;
      for i in Store'range LOOP
         Store(i) := Arg1(i) nor Arg2(i);
      end LOOP;
      return STORE;
   end "nor";

-------------------------------------------------------------------------
FUNCTION "xor" ( i1, i2 : logic_vector_mv ) RETURN logic_vector_mv is
   alias Arg1 : logic_vector_mv ( 1 to i1'length ) is i1;
   alias Arg2 : logic_vector_mv ( 1 to i2'length ) is i2;
   variable Store : logic_Vector_mv ( 1 to i1'length );

   begin
      assert i1'length = i2'length report "Bus width Mismatch! "
                                          severity warning;
```

```
      for i in Store'range LOOP
         Store(i) := Arg1(i) xor Arg2(i);
      end LOOP;
      return STORE;
   end "xor";

-------------------------------------------------------------------------
FUNCTION xnor ( i1, i2 : logic_vector_mv ) RETURN logic_vector_mv is
   alias Arg1 : logic_vector_mv ( 1 to i1'length ) is i1;
   alias Arg2 : logic_vector_mv ( 1 to i2'length ) is i2;
   variable Store : logic_Vector_mv ( 1 to i1'length );

   begin
      assert i1'length = i2'length report "Bus width Mismatch! "
                                    severity warning;
      for i in Store'range LOOP
         Store(i) := xnor ( Arg1(i), Arg2(i) );
      end LOOP;
      return STORE;
   end xnor;

-------------------------------------------------------------------------

FUNCTION "not" ( i1 : logic_vector_mv ) RETURN logic_vector_mv is
   variable Store : logic_Vector_mv ( 1 to i1'length );

   begin
      for i in Store'range LOOP
         Store(i) := not i1(i);
      end LOOP;
      return STORE;
   end "not";


-------------------------------------------------------------------------
--  BIT-WISE REDUCTION OPERATORS
-------------------------------------------------------------------------
FUNCTION and_bw ( i1 : logic_vector_mv )  RETURN logic_mv is
   variable Store : logic_mv := i1(i1'low);

   begin
      for i in i1'low + 1 to i1'high LOOP
         CASE Store is
            when ZERO | LOW => RETURN ZERO ;
            when ONE | HIGH =>   Case i1(i) is
                                    when ZERO | LOW => RETURN ZERO ;
                                    when ONE | HIGH => NULL;
                                    when others => Store := UNKNOWN;
                                 end CASE;
            when others =>       Case i1(i) is
                                    when ZERO | LOW => RETURN ZERO ;
                                    when others => Store := UNKNOWN;
                                 end CASE;
         end CASE;
      end LOOP;
      RETURN Filter (Store);
   end and_bw;
-------------------------------------------------------------------------
```

```
FUNCTION nand_bw ( i1 : logic_vector_mv )  RETURN logic_mv is
   variable Store : logic_mv := i1(i1'low);

   begin
      for i in i1'low + 1 to i1'high LOOP
         CASE Store is
            when ZERO | LOW => RETURN ONE;
            when ONE | HIGH =>   Case i1(i) is
                                    when ZERO | LOW => RETURN ONE;
                                    when ONE | HIGH => NULL;
                                    when others => Store := UNKNOWN;
                                 end CASE;
            when others =>       Case i1(i) is
                                    when ZERO | LOW => RETURN ONE;
                                    when others => Store := UNKNOWN;
                                 end CASE;
         end CASE;
      end LOOP;
      RETURN not Store;
   end nand_bw;
-------------------------------------------------------------------------
FUNCTION or_bw ( i1 : logic_vector_mv )  RETURN logic_mv is
   variable Store : logic_mv := i1(i1'low);

   begin
      for i in i1'low + 1 to i1'high LOOP
         CASE Store is
            when ONE | HIGH => RETURN ONE;
            when ZERO | LOW =>   Case i1(i) is
                                    when ZERO | LOW => NULL;
                                    when ONE | HIGH => RETURN ONE;
                                    when others => Store := UNKNOWN;
                                 end CASE;
            when others =>       Case i1(i) is
                                    when ONE | HIGH => RETURN ONE;
                                    when others => Store := UNKNOWN;
                                 end CASE;
         end CASE;
      end LOOP;
      RETURN Filter(Store);
   end or_bw;
-------------------------------------------------------------------------
FUNCTION nor_bw ( i1 : logic_vector_mv )  RETURN logic_mv is
   variable Store : logic_mv := i1(i1'low);

   begin
      for i in i1'low + 1 to i1'high LOOP
         CASE Store is
            when ONE | HIGH => RETURN ZERO;
            when ZERO | LOW =>   Case i1(i) is
                                    when ZERO | LOW => NULL;
                                    when ONE | HIGH => RETURN ZERO;
                                    when others => Store := UNKNOWN;
                                 end CASE;
            when others =>       Case i1(i) is
                                    when ONE | HIGH => RETURN ZERO;
                                    when others => Store := UNKNOWN;
                                 end CASE;
```

```
         end CASE;
      end LOOP;
      RETURN not Store;
   end nor_bw;
-------------------------------------------------------------------------
FUNCTION xor_bw ( i1 : logic_vector_mv )  RETURN logic_mv is
   variable Store : logic_mv := i1(i1'low);

   begin
      IF i1'length > 1 then
         for i in i1'low + 1 to i1'high LOOP
            CASE Store is
               when ZERO | LOW =>   Case i1(i) is
                                       when ZERO | LOW => NULL;
                                       when ONE | HIGH => RETURN ONE;
                                       when others => RETURN UNKNOWN;
                                    end CASE;
               when ONE | HIGH =>         Case i1(i) is
                                       when ZERO | LOW => RETURN ONE;
                                       when ONE | HIGH => NULL;
                                       when others => RETURN UNKNOWN;
                                    end CASE;
               when others =>     RETURN UNKNOWN;
            end CASE;
         end LOOP;
         RETURN ZERO;
      else
         RETURN Filter(Store);
      end if;
   end xor_bw;
-------------------------------------------------------------------------
FUNCTION xnor_bw ( i1 : logic_vector_mv )  RETURN logic_mv is
   variable Store : logic_mv := i1(i1'low);

   begin
      IF i1'length > 1 then
         for i in i1'low + 1 to i1'high LOOP
            CASE Store is
               when ZERO | LOW =>   Case i1(i) is
                                       when ZERO | LOW => NULL;
                                       when ONE | HIGH => RETURN ZERO;
                                       when others => RETURN UNKNOWN;
                                    end CASE;
               when ONE | HIGH =>         Case i1(i) is
                                       when ZERO | LOW => RETURN ZERO;
                                       when ONE | HIGH => NULL;
                                       when others => RETURN UNKNOWN;
                                    end CASE;
               when others =>     RETURN UNKNOWN;
            end CASE;
         end LOOP;
         RETURN ONE;
      else
         RETURN not Store;
      end if;
   end xnor_bw;

-------------------------------------------------------------------------
```

```
--  COMPARISON OPERATORS
-------------------------------------------------------------------------
FUNCTION "=" ( i1, i2 : logic_mv ) RETURN logic_mv is
   constant table : logic_mv_table :=
   (( UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN),
   (  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN),
   (  UNKNOWN,  UNKNOWN,  ONE,      ZERO,     UNKNOWN,  ONE,      ZERO),
   (  UNKNOWN,  UNKNOWN,  ZERO,     ONE,      UNKNOWN,  ZERO,     ONE),
   (  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN),
   (  UNKNOWN,  UNKNOWN,  ONE,      ZERO,     UNKNOWN,  ONE,      ZERO),
   (  UNKNOWN,  UNKNOWN,  ZERO,     ONE,      UNKNOWN,  ZERO,     ONE));
   begin
      RETURN table (i1, i2);
   end "=";

-------------------------------------------------------------------------
FUNCTION "/=" ( i1, i2 : logic_mv ) RETURN logic_mv is
   constant table : logic_mv_table :=
   (( UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN),
   (  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN),
   (  UNKNOWN,  UNKNOWN,  ZERO,     ONE,      UNKNOWN,  ZERO,     ONE),
   (  UNKNOWN,  UNKNOWN,  ONE,      ZERO,     UNKNOWN,  ONE,      ZERO),
   (  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN,  UNKNOWN),
   (  UNKNOWN,  UNKNOWN,  ZERO,     ONE,      UNKNOWN,  ZERO,     ONE),
   (  UNKNOWN,  UNKNOWN,  ONE,      ZERO,     UNKNOWN,  ONE,      ZERO));
   begin
      RETURN table (i1, i2);
   end "/=";

-------------------------------------------------------------------------
FUNCTION "=" ( i1, i2 : logic_vector_mv ) RETURN logic_mv is
   alias Arg1 : logic_vector_mv ( 1 to i1'length ) is i1;
   alias Arg2 : logic_vector_mv ( 1 to i2'length ) is i2;
   variable Store : logic_mv;

   begin
      assert i1'length = i2'length report "Bus Width Mismatch!"
         severity warning;
      for i in i1'range LOOP
         Store := Arg1(i) = Arg2(i);
         if Store /= ONE then
            RETURN Store;
         end if;
      end LOOP;
   RETURN ONE;
   end "=";

-------------------------------------------------------------------------
FUNCTION "/=" ( i1, i2 : logic_vector_mv ) RETURN logic_mv is
   alias Arg1 : logic_vector_mv ( 1 to i1'length ) is i1;
   alias Arg2 : logic_vector_mv ( 1 to i2'length ) is i2;
   variable Store : logic_mv;

   begin
      assert i1'length = i2'length report "Bus Width Mismatch!"
         severity warning;
      for i in i1'range LOOP
         Store := Arg1(i) /= Arg2(i);
```

```
         if Store /= ONE then
            RETURN Store;
         end if;
      end LOOP;
   RETURN ONE;
   end "/=";

-------------------------------------------------------------------------
BUS RESOLUTION FUNCTIONS
-------------------------------------------------------------------------
FUNCTION Wired_Outputs ( signals : logic_vector_mv ) RETURN logic_mv is
   variable result : logic_mv := HIGHZ;  -- return 'Z' when no active
driver
   constant Table : logic_mv_table :=
   (( UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN ),
   (  UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN ),
   (  UNKNOWN, UNKNOWN, ZERO,    UNKNOWN, ZERO,    ZERO,    ZERO ),
   (  UNKNOWN, UNKNOWN, UNKNOWN, ONE,     ONE,     ONE,     ONE ),
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     HIGHZ,   LOW,     HIGH ),
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     LOW,     LOW,     HIGHZ ),
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     HIGH,    HIGHZ,   HIGH ));
   begin
      for i in signals'range LOOP
               result := table ( result, signals(i) );
               exit when result = UNKNOWN;
      end LOOP;
      return result;
   end Wired_Outputs;

-------------------------------------------------------------------------
FUNCTION Wired_Or ( signals : logic_vector_mv ) RETURN logic_mv is
   variable result : logic_mv := HIGHZ;  -- return 'Z' when no active
driver
   constant Table : logic_mv_table :=
   (( UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN ),
   (  UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN ),
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     ZERO,    ZERO,    ZERO ),
   (  UNKNOWN, UNKNOWN, ONE,     ONE,     ONE,     ONE,     ONE ),
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     HIGHZ,   LOW,     HIGH ),
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     LOW,     LOW,     HIGHZ ),
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     HIGH,    HIGHZ,   HIGH ));
   begin
      for i in signals'range LOOP
               result := table ( result, signals(i) );
               exit when result = UNKNOWN;
      end LOOP;
      return result;
   end Wired_Or;

-------------------------------------------------------------------------
FUNCTION Wired_AND ( signals : logic_vector_mv ) RETURN logic_mv is
   variable result : logic_mv := HIGHZ;  -- return 'Z' when no active
driver
   constant Table : logic_mv_table :=
   (( UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN ),
   (  UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN, UNKNOWN ),
   (  UNKNOWN, UNKNOWN, ZERO,    ZERO,    ZERO,    ZERO,    ZERO ),
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     ONE,     ONE,     ONE ),
```

```
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     HIGHZ,   LOW,     HIGH ),
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     LOW,     LOW,     HIGHZ ),
   (  UNKNOWN, UNKNOWN, ZERO,    ONE,     HIGH,    HIGHZ,   HIGH ));
   begin
      for i in signals'range LOOP
               result := table ( result, signals(i) );
               exit when result = UNKNOWN;
      end LOOP;
      return result;
   end Wired_AND;

-------------------------------------------------------------------------
--  Miscellaneous Functions
-------------------------------------------------------------------------
--
--    FUNCTION name : Filter
--                translates logic_mv states:
--                      HIGH -> ONE
--                      LOW -> ZERO
--                      HIGHZ -> UNKNOWN
--
FUNCTION Filter ( input : logic_mv ) RETURN logic_mv is
      constant filter_table : logic_mv_array :=
      ( UNKNOWN,  UNKNOWN,  ZERO,  ONE,  UNKNOWN,  ZERO, ONE);
      begin
                RETURN filter_table( input );
      end Filter;

-------------------------------------------------------------------------
Signal Transitions and Relationships
-------------------------------------------------------------------------
FUNCTION Posedge ( signal s1 : logic_mv ) RETURN boolean is
   begin
      RETURN s1 = ONE and s1'last_value = ZERO and s1'event;
   end Posedge;

-------------------------------------------------------------------------
FUNCTION Negedge ( signal s1 : logic_mv ) RETURN boolean is
   begin
      RETURN s1 = ZERO and s1'last_value = ONE and s1'event;
   end negedge;

-------------------------------------------------------------------------
PROCEDURE Setup_check (    constant input_le : time;
            constant time_spec : time;
            constant message : string;
            constant err_level : severity_level) is
   begin
      assert input_le >= time_spec
         report message & " setup violation" severity err_level;
   end Setup_check;

-------------------------------------------------------------------------
PROCEDURE Hold_check (  constant input_le       : time;
                        constant time_spec      : time;
                        constant message        : string;
                        constant err_level      : severity_level) is
   begin
```

```
      assert input_le > time_spec
            report message & " setup violation" severity err_level;
   end Hold_check;

-------------------------------------------------------------------------
FUNCTION F_delay ( newlv : in logic_mv;
         delay01 : in time;
         delay10 : in time)  RETURN time is
   begin
      CASE newlv is
         when ZERO =>   RETURN      delay10;
         when ONE  =>   RETURN      delay01;
         when others => if ( delay01 > delay10 ) then return delay01;
                        else
                           return delay10;
                        end if;
      end CASE;
   end F_delay;


end BASICDEFS;
```

Appendix B: The Behavioral Model

This appendix contains complete behavioral examples of sequential circuits specified via the model proposed in Chapter 3. Each example is presented in the following order:

1. Packages and package bodies required by the design,
2. The sequential circuit's entity description,
3. The sequential circuit's architectural body,
4. The test bench entity and architectural body used to test the sequential circuit, and,
5. A simulation report generated from a sample run.

The first example is a synchronous Mealy sequential circuit. The second example is an asynchronous Moore circuit. The final example is the CPU controller (a synchronous "hybrid" sequential circuit). They may be found on the following pages:

```
------------------------------------------------------------------
--
-- File name:      Sequential_Circuit_Package.vhd
--
-- Description:    The following VHDL code is a draft version of
--                 the State_machine package required to support
--                 the various architectural models.
--
-- Status:  It is complete.
--
--
-- Support files: BASICDEFS.vhd      (EIA's BASICDEF's package)
--
-- Creation Date: 11 July 90
--
-- Created by:     Rick Miller
--    Address:     AFIT/ENG
--                 Wright-Patterson AFB, OH, 45433
--
--    Phone:       (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------

------------------------------------------------------------------
-- Package declarations
------------------------------------------------------------------
use work.BASICDEFS.all;
      -- the EIA Standard package BASICDEFS

package Sequential_Circuit_Package is


      Type States is (
                  Unknown_State,
                  First_State,
                  Second_State,
                  Third_State
                  );


      constant    First_State_output       : logic_vector_mv := "001";
      constant    Second_State_output      : logic_vector_mv := "010";
      constant    Third_State_output       : logic_vector_mv := "100";


      Type Transition_Conditions is (
                        No_Transition,
                        goto_First_State,
                        goto_Second_State,
                        goto_Third_State
                        );

      Type Transition_Conditions_vector is array (natural range <>) of
Transition_Conditions;
```

```
-------------------------------------------------------------------
--  Transition Resolution Function
-------------------------------------------------------------------

      Function Transition_Resolution (i1 : Transition_Conditions_vector)
return Transition_Conditions;

end Sequential_Circuit_Package;




package body Sequential_Circuit_Package is

-------------------------------------------------------------------
--  Transition Resolution Function
-------------------------------------------------------------------

      Function Transition_Resolution (i1 : Transition_Conditions_vector)
                  return Transition_Conditions is

      begin
            for I in i1'Range loop
                  RETURN i1(I);

            end loop;
            RETURN No_Transition;
      end;

end Sequential_Circuit_Package ;
```

```
------------------------------------------------------------------
--
-- File name:      Sequential_Circuit_.vhd
--
-- Description:    synchronous Mealy state machine entity.
--
-- Status:  It is complete.
--
-- Support files:        BASICDEFS.vhd      (EIA's BASICDEFs package)
--
-- Creation Date 11 July 90
--
-- Created by:     Rick Miller
--    Address:     AFIT/ENG
--                 Wright-Patterson AFB, OH, 45433
--
--     Phone:  (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------
--  Entity: Sequential_Circuit
------------------------------------------------------------------

use work.Sequential_Circuit_Package .all;
      -- This package makes the appropriate type and variable
      -- declarations required by the particular sequential circuit.

use work.BASICDEFS.all;
      -- The BASICDEFS package is pressumed located in a VHDL design
      -- library sublibrary named EIA.


entity Sequential_Circuit is
      -- generic();

      port(
                  input_signals     : in logic_vector_mv (1 downto 0);
                  out_1,
                  out_2,
                  out_3       : out  logic_mv := 'U';
                  RESET       : in logic_mv;
                  CLOCK       : in logic_mv

            );

end Sequential_Circuit ;
```

```
------------------------------------------------------------------
--
-- File name:     Sequential_Circuit.vhd
--
-- Description:   synchronous Mealy state machine architecture.
--
-- Status:  It is complete.
--
-- Support files:       BASICDEFS.vhd     (EIA's BASICDEFs package)
--                      Sequential_Circuit_Package.vhd
--
-- Creation Date 11 July 90
--
-- Created by:    Rick Miller
--    Address:    AFIT/ENG
--                Wright-Patterson AFB, OH, 45433
--
--     Phone:  (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------
--  Architecture: Sequential_Circuit
------------------------------------------------------------------
architecture synch_mealy_arch of Sequential_Circuit is


      signal  Present_State, Next_State : States := Unknown_State;

      signal      Transition :      Transition_Resolution
                                    Transition_Conditions
                                    BUS := No_Transition;

      signal      state_out_1,
                  state_out_2,
                  state_out_3   : Wired_Outputs logic_mv BUS := 'U';

begin
                        out_1 <= state_out_1;
                        out_2 <= state_out_2;
                        out_3 <= state_out_3;

-- synchronize the state to state transitions to the clock.
process (clock)
      begin
            if (clock = '1' and clock'event)
                  then Present_State <= Next_State;

            end if;
      end process;

-- initialize and reset capability
--  First State Process
RESET_BLOCK:block (RESET = '1')
      begin
      process (GUARD) begin
            if  GUARD then Transition <= goto_First_State;
            else
                  Transition <= null;
```

```
            end if;
      end process;
      end block RESET_BLOCK;

-- the State Transitions
process (transition)
      begin
      case Present_State is
            when First_State =>
                  case transition is
                        when goto_second_State =>
                              Next_State <= Second_State;
                        when goto_Third_State =>
                              Next_State <= Third_State;
                        when others =>
                  end case;

            when  Second_State =>
                  case transition is
                        when goto_First_state =>
                              Next_State <= First_State;
                        when goto_Third_state =>
                              Next_State <= Third_State;
                        when others =>
                  end case;

            when  Third_State =>
                  case transition is
                        when goto_First_state =>
                              Next_State <= First_State;
                        when goto_Second_state =>
                              Next_State <= Second_State;
                        when others =>
                  end case;

            when Unknown_State =>
                  case transition is
                        when goto_First_state =>
                              Next_State <= First_State;
                        when others =>
                  end case;

      end case;
end process;


--  First State Process
FIRST:block (Present_State = First_State)
      begin
      process (GUARD, INPUT_signals) begin
      if GUARD then
            case INPUT_signals is
                  when "10" => Transition <=  goto_Second_State;
                        state_out_1 <=     Second_State_Output(0);
                        state_out_2 <=     Second_State_Output(1);
                        state_out_3 <=     Second_State_Output(2);
```

```
                  when "11" => Transition <=  goto_Third_State;
                        state_out_1 <=     Third_State_Output(0);
                        state_out_2 <=     Third_State_Output(1);
                        state_out_3 <=     Third_State_Output(2);

                  when others =>
                        transition <= no_transition;
                        state_out_1 <= '0';
                        state_out_2 <= '0';
                        state_out_3 <= '0';

            end case;
      else
            transition <= null;
            state_out_1 <= null;
            state_out_2 <= null;
            state_out_3 <= null;
      end if;
      end process;
      end block FIRST;

--  Second State Process
SECOND:block (Present_State = Second_State)
      begin
      process (GUARD, INPUT_signals) begin
      if GUARD then
            case INPUT_signals is
                  when "01" => Transition <= goto_First_State;
                        state_out_1 <=     First_State_Output(0);
                        state_out_2 <=     First_State_Output(1);
                        state_out_3 <=     First_State_Output(2);

                  when "11" => Transition <= goto_Third_State;
                        state_out_1 <=     Third_State_Output(0);
                        state_out_2 <=     Third_State_Output(1);
                        state_out_3 <=     Third_State_Output(2);

                  when others =>
                        Transition <= no_transition;
                        state_out_1 <= '0';
                        state_out_2 <= '0';
                        state_out_3 <= '0';

            end case;
      else
            transition  <= null;
            state_out_1 <= null;
            state_out_2 <= null;
            state_out_3 <= null;
      end if;
      end process;
      end block SECOND;

--  Third State Process
THIRD:block (Present_State = Third_State)
      begin
      process (GUARD, INPUT_signals) begin
```

```
        if GUARD then
              case INPUT_signals is
                    when "01" => Transition <= goto_First_State;
                          state_out_1 <=     First_State_Output(0);
                          state_out_2 <=     First_State_Output(1);
                          state_out_3 <=     First_State_Output(2);

                    when "10" => Transition <= goto_Second_State;
                          state_out_1 <=     Second_State_Output(0);
                          state_out_2 <=     Second_State_Output(1);
                          state_out_3 <=     Second_State_Output(2);

                    when others =>
                          transition  <= NO_TRANSITION;
                          state_out_1 <= '0';
                          state_out_2 <= '0';
                          state_out_3 <= '0';

              end case;
        else
              transition  <= null ;
              state_out_1 <= null;
              state_out_2 <= null;
              state_out_3 <= null;
        end if;
        end process;
        end block THIRD;

end synch_mealy_arch;
```

```
------------------------------------------------------------------
--
-- File name:      testbench.vhd
--
-- Description:    testbench for synchronous Mealy state machine
--                 architecture.
--
-- Status:  It is complete.
--
-- Support files:        BASICDEFS.vhd     (EIA's BASICDEFs package)
--                       Sequential_Circuit_Package.vhd
--
-- Creation Date 11 July 90
--
-- Created by:     Rick Miller
--    Address:     AFIT/ENG
--                 Wright-Patterson AFB, OH, 45433
--
--     Phone:  (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------
--  Architecture: Sequential_Circuit
------------------------------------------------------------------
use work.BASICDEFS.all;
use WORK.Sequential_Circuit_Package.all;
use STD.SIMULATOR_STANDARD.all;
use STD.TEXTIO.all;

entity TEST_BENCH is
end TEST_BENCH;

architecture synch_Mealy_example of TEST_BENCH is

  component state_machine
            port(
                  input_signals     : in logic_vector_mv (1 downto 0);
                  out_1,
                  out_2,
                  out_3       : out  logic_mv;
                  RESET       : in logic_mv;
                  CLOCK       : in logic_mv
                  );

  end component;

  for all : Sequential_Circuit use
            entity WORK.Sequential_Circuit (synch_mealy_arch);

  signal instruction    :     logic_vector_mv (1 downto 0) := "00";
  signal CLOCK          :     logic_mv;
  signal RESET          :     logic_mv;
  signal MOORE_STATE    :     STATEs;
  signal    OUT_1,
            OUT_2,
            OUT_3       :     logic_mv;

begin
```

```
  RESET <= '1' , '0' after 5ns;

process
    file INSTRUCTIONS  : TEXT is in "MEALY_INSTRUCTIONS";
    variable L      : Line;
    variable machine_code : bit_vector (1 downto 0);
    begin
      readline(INSTRUCTIONS, L);
            if ENDFILE(INSTRUCTIONS) then terminate; end if;
      read(L, machine_code);
            case machine_code is
                  when "00" => instruction <= "00";
                  when "01" => instruction <= "01";
                  when "10" => instruction <= "10";
                  when "11" => instruction <= "11";
            end case;
      wait for 30ns;
    end process;



 process
    begin
      set_maximums(10000,100);
      tracing_on;
      wait for 200ns;
      terminate;
    end process;


  make_Clock : process
    begin
      wait for 2ns;
       CLOCK <= '1';
       wait for 4ns;
       CLOCK <= '0';
      wait for 2ns;
    end process make_Clock;

  UUT : Sequential_Circuit
      port map ( instruction,
                OUT_1, OUT_2, OUT_3,
                RESET,
                CLOCK
              );


end synch_Mealy_example;
```

Synch_Mealy_arch"

Vhdl Simulation Report

Report Name: Synch_Mealy_arch"
Kernel Library Name: <<RMILLER.WORKING_EXAMPLES.SYNCH_MEALY>>SYNCH_MEALY_EXAMPLE
Kernel Creation Date: AUG-08-1990
Kernel Creation Time: 23:29:43
Run Identifer: 1
Run Date: AUG-08-1990
Run Time: 23:29:43

Report Control Language File: MEALY.rcl
Report Output File : synch_mealy_example.rpt

Max Time: 9223372036854775807
Max Delta: 2147483646

Report Control Language :

```
Simulation_report State_Machine is
begin
report_name is "Synch_Mealy_arch";
page_width is 120;
page_length is 40;
signal_format is horizontal;
sample_signals by_transaction in ns;
--sample_signals by_event in ns;
select_signal : Clock;
select_signal : reset;
select_signal : instruction;
select_signal : out_1;
select_signal : out_2;
select_signal : out_3;
select_signal /uut : transition;
select_signal  /uut : Present_STATE;
end State_Machine;
```

Report Format Information :

Time is in NS relative to the start of simulation
Time period for report is from 0 NS to End of Simulation
Signal values are reported by transaction (' ' indicates no transaction)

TIME	SIGNAL NAMES							
(NS)	CLOCK	RESET	INSTRUCTION(1 DOWNTO 0)	OUT_1	OUT_2	OUT_3	TRANSITION	PRESENT_STATE
0	'U'	'U'	"00"	'U'	'U'	'U'	NO_TRANSITION	UNKNOWN_STATE
+1		'1'	"01"	'X'	'X'	'X'	NO_TRANSITION	
+2				'Z'	'Z'	'Z'	GOTO_FIRST_STATE	
2								
+1	'1'							
+2								FIRST_STATE
+3							GOTO_FIRST_STATE	
+4				'0'	'0'	'0'		
5		'0'						
+1							NO_TRANSITION	
6								
+1	'0'							
10								
+1	'1'							
+2								FIRST_STATE
14								
+1	'0'							
18								
+1	'1'							
+2								FIRST_STATE
22								
+1	'0'							
26								
+1	'1'							
+2								FIRST_STATE
30								
+1	'0'		"10"					
+2							GOTO_SECOND_STATE	
+3					'1'			
34								
+1	'1'							
+2								SECOND_STATE

Synch_Mealy_arch"

TIME	SIGNAL NAMES							
(NS)	CLOCK	RESET	INSTRUCTION(1 DOWNTO 0)	OUT_1	OUT_2	OUT_3	TRANSITION	PRESENT_STATE
+3							NO_TRANSITION	
+4					'0'			
38								
+1	'0'							
42								
+1	'1'							
+2								SECOND_STATE
46								
+1	'0'							
50								
+1	'1'							
+2								SECOND_STATE
54								
+1	'0'							
58								
+1	'1'							
+2								SECOND_STATE
60								
+1			"11"					
+2							GOTO_THIRD_STATE	
+3				'1'				
62								
+1	'0'							
66								
+1	'1'							
+2								THIRD_STATE
+3							NO_TRANSITION	
+4				'0'				
70								
+1	'0'							
74								
+1	'1'							

Synch_Mealy_arch"

TIME (NS)	CLOCK	RESET	INSTRUCTION(1 DOWNTO 0)	OUT_1	OUT_2	OUT_3	TRANSITION	PRESENT_STATE
+2								THIRD_STATE
78								
+1	'0'							
82								
+1	'1'							
+2								THIRD_STATE
86								
+1	'0'							
90								
+1	'1'		"10"					
+2							GOTO_SECOND_STATE	THIRD_STATE
+3					'1'			
94								
+1	'0'							
98								
+1	'1'							
+2								SECOND_STATE
+3							NO_TRANSITION	
+4					'0'			
102								
+1	'0'							
106								
+1	'1'							
+2								SECOND_STATE
110								
+1	'0'							
114								
+1	'1'							
+2								SECOND_STATE
118								
+1	'0'							
120								

TIME	SIGNAL NAMES							
(NS)	CLOCK	RESET	INSTRUCTION(1 DOWNTO 0)	OUT_1	OUT_2	OUT_3	TRANSITION	PRESENT_STATE
+1			"11"					
+2							GOTO_THIRD_STATE	
+3				'1'				
122								
+1	'1'							
+2								THIRD_STATE
+3							NO_TRANSITION	
+4				'0'				
126								
+1	'0'							
130								
+1	'1'							
+2								THIRD_STATE
134								
+1	'0'							
138								
+1	'1'							
+2								THIRD_STATE
142								
+1	'0'							
146								
+1	'1'							
+2								THIRD_STATE
150								
+1	'0'		"10"					
+2							GOTO_SECOND_STATE	
+3					'1'			
154								
+1	'1'							
+2								SECOND_STATE
+3							NO_TRANSITION	
+4					'0'			

Synch_Mealy_arch"

TIME	SIGNAL NAMES							
(NS)	CLOCK	RESET	INSTRUCTION(1 DOWNTO 0)	OUT_1	OUT_2	OUT_3	TRANSITION	PRESENT_STATE
158								
+1	'0'							
162								
+1	'1'							
+2								SECOND_STATE
166								
+1	'0'							
170								
+1	'1'							
+2								SECOND_STATE
174								
+1	'0'							
178								
+1	'1'							
+2								SECOND_STATE
180								
+1			"01"					
+2							GOTO_FIRST_STATE	
+3						'1'		
182								
+1	'0'							
186								
+1	'1'							
+2								FIRST_STATE
+3							NO_TRANSITION	
+4						'0'		
190								
+1	'0'							
194								
+1	'1'							
+2								FIRST_STATE
198								

Synch_Mealy_arch"

TIME	SIGNAL NAMES								
(NS)	CLOCK	RESET	INSTRUCTION(1 DOWNTO 0)	OUT_1	OUT_2	OUT_3	TRANSITION	PRESENT_STATE	
+1	'0'								

```
------------------------------------------------------------------
-- PACKAGE:  State_Machine_Package
------------------------------------------------------------------
--
-- File name:      moore_pkg.vhd
--
-- Description:    The following VHDL code is a draft version of
--                 the State_machine package required to support
--                 the various architectural models.
--
-- Status:  It is complete and has been successfully simulated.
--
--
-- Support files: EIA_basicdefs.vhd (EIA's BASICDEF's package)
--
-- Creation Date: 11 July 90
--
-- Created by:     Rick Miller
--    Address:     AFIT/ENG
--                 Wright-Patterson AFB, OH, 45433
--
--    Phone:       (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------

use work.BASICDEFS.all;
      -- the EIA Standard package BASICDEFS

package State_Machine_Package is


      Type States is (
                  Unknown_State,
                  First_State,
                  Second_State,
                  Third_State
                  );


      constant    First_State_output       : logic_vector_mv := "001";
      constant    Second_State_output      : logic_vector_mv := "010";
      constant    Third_State_output       : logic_vector_mv := "100";


      Type Transition_Conditions is (
                        No_Transition,
                        goto_First_State,
                        goto_Second_State,
                        goto_Third_State
                        );

      Type Transition_Conditions_vector is array (natural range <>) of
Transition_Conditions;



------------------------------------------------------------------
```

```
--  Transition Resolution Function
-------------------------------------------------------------------

      Function Transition_Resolution (il : Transition_Conditions_vector)
return Transition_Conditions;

end State_Machine_Package;
```

```
------------------------------------------------------------------
--  PACKAGE BODY for State_Machine_Package
------------------------------------------------------------------
--
-- File name:      moore_pkg_body.vhd
--
-- Description:    The following VHDL code is a draft version of the
--                 generic state machine entity.
--
-- Status:  It is complete and has been successfully simulated.
--
-- Support files:        BASICDEFS.vhd      (EIA's BASICDEFs package)
--
-- Creation Date 11 July 90
--
-- Created by:     Rick Miller
--    Address:     AFIT/ENG
--                 Wright-Patterson AFB, OH, 45433
--
--     Phone:  (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------

package body State_Machine_Package is

------------------------------------------------------------------
--  Transition Resolution Function
------------------------------------------------------------------

      Function Transition_Resolution (i1 : Transition_Conditions_vector)
                  return Transition_Conditions is

      begin
            for I in i1'Range loop
                  RETURN i1(I);

            end loop;
            RETURN No_Transition;
      end;

end State_Machine_Package;
```

```
------------------------------------------------------------------
--  Entity: State_Machine for asynchronous moore example
------------------------------------------------------------------
--
-- File name:      state_machine_.vhd
--
-- Description:    The following VHDL code is a draft version of the
--                 generic state machine entity.
--
-- Status:  It is complete and has been successfully simulated.
--
-- Support files:        BASICDEFS.vhd     (EIA's BASICDEFs package)
--                 MOORE_pkg.vhd     (State_machine_packagae)
--                 MOORE_pkg_body.vhd (body of above)
--                 testbench.vhd
--
-- Creation Date 11 July 90
--
-- Created by: Rick Miller
--    Address:     AFIT/ENG
--                 Wright-Patterson AFB, OH, 45433
--
--     Phone:  (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------
use work.State_Machine_Package.all;
      -- This package makes the appropriate type and variable
declarations
      -- required by the particular state machine.

use work.BASICDEFS.all;
      -- The BASICDEFS package is pressumed located in a VHDL design
      -- library sublibrary named EIA.


entity State_Machine is
      -- generic();

      port(
                  input_signals     : in logic_vector_mv (1 downto 0);
                  out_1,
                  out_2,
                  out_3       : out  logic_mv := 'U';
                  RESET       : in logic_mv

            );

end State_Machine;
```

```
------------------------------------------------------------------
--  ARCHITECTURE: Asynchronous Moore for entity State_Machine
------------------------------------------------------------------
--
-- File name:      synch_moore_arch.vhd
--
-- Description:   The following VHDL code is a draft version of the
--          asynchronous moore state machine.
--
-- Status:  It is complete and has been successfully simulated.
--
-- Support files: BASICDEFS.vhd      (EIA's BASICDEFs package)
--                MOORE_pkg.vhd      (State_machine_packagae)
--                MOORE_pkg_body.vhd (body of above)
--                asynch_moore_entity.vhd
--                testbench.vhd
--
-- Creation Date 11 July 90
--
-- Created by:    Rick Miller
--    Address:    AFIT/ENG
--                Wright-Patterson AFB, OH, 45433
--
--     Phone:  (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------

architecture Asynch_Moore of State_Machine is

------------------------------------------------------------------
--    Declarative block
--          The types States and Transition_Conditions are found
--          in the State_Machine_Package.  (Moore_pkg.vhd)
--
--          The type logic_mv is found in the package BASICDEFS.
--          (EIA_basicdefs.vhd)
--
--          The resolution function Transition_Resolution is
--          found in the State_Machine_Package.  (Moore_pkg.vhd)
--
--          The resolution function Wired_Outputs is found in
--          the package BASICDEFS.  (EIA_basicdefs.vhd)
--
------------------------------------------------------------------

      signal  Present_State, Next_State : States := Unknown_State;

      signal      Transition :      Transition_Resolution
                                    Transition_Conditions
                                    BUS := No_Transition;

      signal      state_out_1,
                  state_out_2,
                  state_out_3  : Wired_Outputs logic_mv BUS := 'U';

begin
```

```
-------------------------------------------------------------------
--    For now, the state outputs are assigned to temporary signals,
--          state_out1, state_out2, and, state_out_3.  These signals
--          are then assigned to the entity's ports here.
--
      out_1 <= state_out_1;
      out_2 <= state_out_2;
      out_3 <= state_out_3;


-------------------------------------------------------------------
--    The following block operates concurently with the state machine.
--          It provides the ability to reset or initialize the
--          machine to a known starting state.

RESET_BLOCK:block (RESET = '1')
      begin
      process (GUARD) begin
            if  GUARD then Transition <= goto_First_State;
            else
                  Transition <= null;
            end if;
      end process;
      end block RESET_BLOCK;

-------------------------------------------------------------------
--    This process determines the "next" state of the state machine.
--    Sensitive to changes in the signal transition, this process only
--    operates when a state to state transition is requested.
--    Optional timing information can be inserted here, such as:
--          Present_State <= SOME_NEW_STATE after some_delay_time;

process (transition)
      begin
      case Present_State is
            when First_State =>
                  case transition is
                        when goto_second_State =>
                              Present_State <= Second_State;
                        when goto_Third_State =>
                              Present_State <= Third_State;
                        when others =>
                  end case;

            when  Second_State =>
                  case transition is
                        when goto_First_state =>
                              Present_State <= First_State;
                        when goto_Third_state =>
                              Present_State <= Third_State;
                        when others =>
                  end case;

            when  Third_State =>
                  case transition is
                        when goto_First_state =>
                              Present_State <= First_State;
```

```
                        when goto_Second_state =>
                              Present_State <= Second_State;
                        when others =>
                  end case;

            when Unknown_State =>
                  case transition is
                        when goto_First_state =>
                              Present_State <= First_State;
                        when others =>
                  end case;

      end case;
end process;

-------------------------------------------------------------------
--    The following blocks represent the individual states of the
--          state machine.  Each block has a guard statement to
--          determine the operation of the block.  That guard signal
--          along with INPUT_signals determines the state operation.
--          If the guard is true, and inputs have changed, then
--          the outputs or the transition variable are set.
--          If the guard is false, null drivers are assigned to the
--          signals.
--          To improve the design's readability, each process may
--          be written to call subprograms which in turn would
--          represent the state's functionality.

--  First State Process
FIRST:block (Present_State = First_State)
      begin
      process (GUARD, INPUT_signals) begin
      if GUARD then
            case INPUT_signals is
                  when "10" => Transition <=  goto_Second_State;
                  when "11" => Transition <=  goto_Third_State;
                  when others =>
                        transition <= no_transition;
                        state_out_1 <= First_State_Output(0);
                        state_out_2 <= First_State_Output(1);
                        state_out_3 <= First_State_Output(2);
            end case;
      else
            transition <= null;
            state_out_1 <= null;
            state_out_2 <= null;
            state_out_3 <= null;
      end if;
      end process;
      end block FIRST;

--  Second State Process
SECOND:block (Present_State = Second_State)
      begin
      process (GUARD, INPUT_signals) begin
      if GUARD then
            case INPUT_signals is
```

```
                  when "01" => Transition <= goto_First_State;
                  when "11" => Transition <= goto_Third_State;
                  when others =>
                        Transition <= no_transition;
                        state_out_1 <= Second_State_Output(0);
                        state_out_2 <= Second_State_Output(1);
                        state_out_3 <= Second_State_Output(2);

            end case;
      else
            transition  <= null;
            state_out_1 <= null;
            state_out_2 <= null;
            state_out_3 <= null;
      end if;
      end process;
      end block SECOND;

--  Third State Process
THIRD:block (Present_State = Third_State)
      begin
      process (GUARD, INPUT_signals) begin
      if GUARD then
            case INPUT_signals is
                  when "01" => Transition <= goto_First_State;
                  when "10" => Transition <= goto_Second_State;
                  when others =>
                        transition  <= NO_TRANSITION;
                        state_out_1 <= Third_State_Output(0);
                        state_out_2 <= Third_State_Output(1);
                        state_out_3 <= Third_State_Output(2);


            end case;
      else
            transition  <= null ;
            state_out_1 <= null;
            state_out_2 <= null;
            state_out_3 <= null;
      end if;
      end process;
      end block THIRD;

end asynch_Moore;
```

```
------------------------------------------------------------------
--  Test Bench description for asynchronous moore example
------------------------------------------------------------------
--
-- File name:      testbench.vhd
--
-- Description:    The following VHDL code provides a test bench
--                 capability for testing the asynchronous moore
--                 state machine.
--
-- Status:  It is complete and has been successfully simulated.
--
-- Support files: BASICDEFS.vhd      (EIA's BASICDEFs package)
--                MOORE_pkg.vhd      (State_machine_packagae)
--                MOORE_pkg_body.vhd (body of above)
--
-- Creation Date 11 July 90
--
-- Created by:     Rick Miller
--    Address:     AFIT/ENG
--                 Wright-Patterson AFB, OH, 45433
--
--     Phone:  (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------

use work.BASICDEFS.all;
use WORK.State_Machine_Package.all;
use STD.SIMULATOR_STANDARD.all;
use STD.TEXTIO.all;

entity TEST_BENCH is
end TEST_BENCH;

architecture asynch_Moore_example of TEST_BENCH is

  component state_machine
            port(
                  input_signals     : in logic_vector_mv (1 downto 0);
                  out_1,
                  out_2,
                  out_3       : out  logic_mv;
                  RESET       : in logic_mv
                  );

  end component;

  for all : state_machine use
            entity WORK.state_machine(asynch_moore);

  signal instruction    :     logic_vector_mv (1 downto 0) := "00";
  signal CLOCK          :     logic_mv;
  signal RESET          :     logic_mv;
  signal MOORE_STATE    :     STATEs;
  signal    OUT_1,
            OUT_2,
            OUT_3       :     logic_mv;
```

```
begin

  RESET <= '1' , '0' after 5ns;

process
    file INSTRUCTIONS  : TEXT is in "MOORE_INSTRUCTIONS";
    variable L     : Line;
    variable machine_code : bit_vector (1 downto 0);
    begin
      readline(INSTRUCTIONS, L);
            if ENDFILE(INSTRUCTIONS) then terminate; end if;
      read(L, machine_code);
            case machine_code is
                  when "00" => instruction <= "00";
                  when "01" => instruction <= "01";
                  when "10" => instruction <= "10";
                  when "11" => instruction <= "11";
            end case;
      wait for 30ns;
    end process;



 process
    begin
      set_maximums(10000,100);
      tracing_on;
      wait for 200ns;
      terminate;
    end process;


  make_Clock : process
    begin
      wait for 2ns;
       CLOCK <= '1';
       wait for 4ns;
       CLOCK <= '0';
      wait for 2ns;
    end process make_Clock;

  UUT : State_Machine
            port map ( instruction,
               OUT_1, OUT_2, OUT_3,
               RESET
             );


end asynch_Moore_example;
```

asynch_Moore_arch"

Vhdl Simulation Report

Report Name: asynch_Moore_arch"
Kernel Library Name: <<RMILLER.WORKING_EXAMPLES.ASYNCH_MOORE>>ASYNCH_MOORE_EXAMPLE
Kernel Creation Date: AUG-08-1990
Kernel Creation Time: 23:58:53
Run Identifer: 1
Run Date: AUG-08-1990
Run Time: 23:58:53

Report Control Language File: MOORE_a.rcl
Report Output File : asynch_moore_example.rpt

Max Time: 9223372036854775807
Max Delta: 2147483646

Report Control Language :

```
-----------------------------------------------------------------
--  REPORT CONTROL FILE for asynchronous moore example
-----------------------------------------------------------------
-- DRAFT  DRAFT  DRAFT  DRAFT  DRAFT  DRAFT  DRAFT  DRAFT  DRAFT
--
-- File name:   MOORE_a.rcl
--
-- Description: The following file is a report control file for
--        the asynchronous Moore Example.
--
-- Status:  It is complete and has been simulated.
--
-- Support files:     None
--
-- Creation Date 11 July 90
--
-- Created by:  Rick Miller
--              Address:AFIT/ENG
--        Wright-Patterson AFB, OH, 45433
```

```
--
--  Phone:      (513)-258-1024  or (513)-255-4960
--
-------------------------------------------------------------------

Simulation_report State_Machine is
begin
report_name is "asynch_Moore_arch";
page_width is 100;
page_length is 40;
signal_format is horizontal;

sample_signals by_transaction in ns;
--sample_signals by_event in ns;
select_signal : reset;
select_signal : instruction;
select_signal : out_1;
select_signal : out_2;
select_signal : out_3;
select_signal /uut : transition;
select_signal  /uut : Present_STATE;

end State_Machine;
```

Report Format Information :

Time is in NS relative to the start of simulation
Time period for report is from 0 NS to End of Simulation
Signal values are reported by transaction (' ' indicates no transaction)

TIME	SIGNAL NAMES						
(NS)	RESET	INSTRUCTION(1 DOWNTO 0)	OUT_1	OUT_2	OUT_3	TRANSITION	PRESENT_STATE
0	'U'	"00"	'U'	'U'	'U'	NO_TRANSITION	UNKNOWN_STATE
+1	'1'	"01"	'X'	'X'	'X'	NO_TRANSITION	
+2			'Z'	'Z'	'Z'	GOTO_FIRST_STATE	
+3							FIRST_STATE
+4						GOTO_FIRST_STATE	
+5			'0'	'0'	'1'		
5	'0'						
+1						NO_TRANSITION	
30							
+1		"10"					
+2						GOTO_SECOND_STATE	
+3							SECOND_STATE
+4						NO_TRANSITION	
+5				'1'	'0'		
60							
+1		"11"					
+2						GOTO_THIRD_STATE	
+3							THIRD_STATE
+4						NO_TRANSITION	
+5			'1'	'0'			
90							
+1		"10"					
+2						GOTO_SECOND_STATE	
+3							SECOND_STATE
+4						NO_TRANSITION	
+5			'0'	'1'			
120							
+1		"11"					
+2						GOTO_THIRD_STATE	
+3							THIRD_STATE
+4						NO_TRANSITION	
+5			'1'	'0'			

TIME	SIGNAL NAMES						
(NS)	RESET	INSTRUCTION(1 DOWNTO 0)	OUT_1	OUT_2	OUT_3	TRANSITION	PRESENT_STATE
150							
+1		"10"					
+2						GOTO_SECOND_STATE	
+3							SECOND_STATE
+4						NO_TRANSITION	
+5			'0'	'1'			
180							
+1		"01"					
+2						GOTO_FIRST_STATE	
+3							FIRST_STATE
+4						NO_TRANSITION	
+5				'0'	'1'		
210							
+1		"10"					
+2						GOTO_SECOND_STATE	
+3							SECOND_STATE
+4						NO_TRANSITION	
+5				'1'	'0'		

```
------------------------------------------------------------------
--
-- File name:      CPU_Package.vhd
--
-- Description:    State, Transition, Constant, and Function package
--                 for the 8 instruction CPU controller.
--
-- Status:  Complete.
--
--
-- Support files: BASICDEFS.vhd      (EIA's BASICDEF's package)
--
-- Creation Date: 1 August 90
--
-- Created by:           Rick Miller
--    Address:           AFIT/ENG
--                       Wright-Patterson AFB, OH, 45433
--
--    Phone:       (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------

------------------------------------------------------------------
-- Package declarations
------------------------------------------------------------------

use WORK.BASICDEFS.all;
      -- the EIA Standard package BASICDEFS

package CPU_Package is


        type STATEs is
                (
                  Unknown_State,
                  AC_gets_AC_plus_DR_State,
                  AC_gets_AC_and_DR_State,
                  AC_gets_NOT_AC_State,
                  READ_M_State,
                  WRITE_M_State,
                  DR_gets_AC_State,
                  AC_gets_DR_State,
                  AR_gets_DR_ADR_State,
                  IR_gets_DR_OP_State,
                  AR_gets_PC_State,
                  RIGHT_SHIFT_AC_State,
                  JUMP_State,
                  READ_INSTRUCTION_State
                );


      Type Transition_Conditions is (
                        No_Transition,
                        goto_RESET,
                        goto_AC_gets_DR,
                        goto_AC_gets_AC_plus_DR,
```

```
                        goto_AC_gets_AC_and_DR,
                        goto_IR_gets_DR_OP,
                        goto_AR_gets_DR_ADR,
                        goto_AR_gets_PC,
                        goto_AC_gets_NOT_AC,
                        goto_RIGHT_SHIFT_AC,
                        goto_Write_M,
                        goto_READ_M,
                        goto_DR_gets_AC    ,
                        goto_READ_INSTRUCTION,
                        goto_JUMP
                        );

      Type Transition_Conditions_vector is array (natural range <>) of
Transition_Conditions;


------------------------------------------------------------------
--  Transition Resolution Function
------------------------------------------------------------------
      Function Transition_Resolution (i1 : Transition_Conditions_vector)
return Transition_Conditions;


      subtype logic_mv_bus is Wired_Outputs logic_mv;
      type logic_vector_mv_bus is array (natural range <>) of
                              logic_mv_bus;
      constant    C0     : logic_vector_mv_bus (12 downto 0)
                              := "0000000000001";
      constant    C1     : logic_vector_mv_bus (12 downto 0)
                              := "0000000000010";
      constant    C2     : logic_vector_mv_bus (12 downto 0)
                              := "0000000000100";
      constant    C3     : logic_vector_mv_bus (12 downto 0)
                              := "0000000001000";
      constant    C4     : logic_vector_mv_bus (12 downto 0)
                              := "0000000010000";
      constant    C5     : logic_vector_mv_bus (12 downto 0)
                              := "0000000100000";
      constant    C6     : logic_vector_mv_bus (12 downto 0)
                              := "0000001000000";
      constant    C7     : logic_vector_mv_bus (12 downto 0)
                              := "0000010000000";
      constant    C8     : logic_vector_mv_bus (12 downto 0)
                              := "0000100000000";
      constant    C9_C3  : logic_vector_mv_bus (12 downto 0)
                              := "0001000001000";
      constant    C10    : logic_vector_mv_bus (12 downto 0)
                              := "0010000000000";
      constant    C11    : logic_vector_mv_bus (12 downto 0)
                              := "0100000000000";
      constant    C12    : logic_vector_mv_bus (12 downto 0)
                              := "1000000000000";
      constant    Uninit       : logic_vector_mv_bus (12 downto 0)
                              := "UUUUUUUUUUUUU";
      constant    zeroes       : logic_vector_mv_bus (12 downto 0)
```

```
                                        := "UUUUUUUUUUUUU";

        Type instructions is
                (
                NOP,
                LOAD,
                STORE,
                ADD,
                BIT_AND,
                JUMP,
                JUMPZ,
                COMP,
                RSHIFT
                );

end CPU_Package;
```

```
package body CPU_Package is

-------------------------------------------------------------------
--  Transition Resolution Function
-------------------------------------------------------------------

      Function Transition_Resolution (i1 : Transition_Conditions_vector)
                  return Transition_Conditions is

      begin
            for I in i1'Range loop
                  RETURN i1(I);

            end loop;
            RETURN No_Transition;
      end;

end CPU_Package;
```

```
------------------------------------------------------------------
--
-- File name:      cpu_controller.vhd
--
-- Description:    Entity and Architecture for 8 instruction cpu
--                 controller
--
-- Status:  Complete.
--
--
-- Support files: BASICDEFS.vhd      (EIA's BASICDEF's package)
--
-- Creation Date: 1 August 90
--
-- Created by:           Rick Miller
--    Address:           AFIT/ENG
--                       Wright-Patterson AFB, OH, 45433
--
--    Phone:       (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------

------------------------------------------------------------------
-- Entity
------------------------------------------------------------------
use work.BASICDEFS.all;
use work.CPU_package.all;

entity CPU_CONTROLLER is
      -- generic ();
      port( instruction : in instructions := NOP ;
            CLOCK       : in logic_mv;
            RESET       : in logic_mv     ;
            ZERO_FLAG   : in logic_mv;
            Control_bus : out logic_vector_mv_bus (12 downto 0)
                        := "XXXXXXXXXXXXX"
            );
end;

------------------------------------------------------------------
-- Architecture
------------------------------------------------------------------
architecture BEHAVIORAL of CPU_CONTROLLER is
      signal  Present_State,
            Next_State  : STATEs    := Unknown_State;
      signal Transition : Transition_Resolution
                              Transition_Conditions
                              BUS
                              := No_Transition;
      signal Control    : logic_vector_mv_bus ( 12 downto 0 )  BUS
                        := "XXXXXXXXXXXXX";

begin
            control_bus <= control;

CLOCK_SYNCH:process (clock) begin
      if posedge( clock ) then Present_State <= Next_State;
```

```
      end if;
      end process CLOCK_SYNCH;

RESET_BLOCK : process (RESET) begin
            if RESET = '1' and RESET'event  then
                  Transition <= goto_RESET;
            else
                  Transition <= null;
            end if;
      end process RESET_BLOCK;


process (Transition) begin

      case Present_State is

            when AR_gets_PC_State =>
                  case Transition is
                        when goto_RESET =>
                              Next_State <= AR_gets_PC_State;
                        when others =>
                              Next_State <= READ_INSTRUCTION_State;
                  end case;

            when READ_M_State =>
                  case Transition is
                        when goto_AC_gets_DR =>
                              NEXT_STATE <= AC_gets_DR_State;

                        when goto_AC_gets_AC_plus_DR=>
                              NEXT_STATE <= AC_gets_AC_plus_DR_State;

                        when goto_AC_gets_AC_and_DR=>
                              NEXT_STATE <= AC_gets_AC_and_DR_State;

                        when goto_RESET =>
                              Next_State <= AR_gets_PC_State;

                        when others  =>
                  end case;

            when READ_INSTRUCTION_State =>
                  case Transition is
                        when goto_IR_gets_DR_OP =>
                              NEXT_STATE <= IR_gets_DR_OP_State;

                        when goto_RESET =>
                              Next_State <= AR_gets_PC_State;

                        when others =>
                  end case;

            when IR_gets_DR_OP_State =>
                  case Transition is
                        when goto_AR_gets_DR_ADR =>
                              NEXT_STATE <= AR_gets_DR_ADR_State;
```

```
            when goto_AR_gets_PC =>
                  NEXT_STATE <= AR_gets_PC_State;

            when goto_JUMP =>
                  NEXT_STATE <= JUMP_State;

            when goto_AC_gets_NOT_AC=>
                  NEXT_STATE <= AC_gets_NOT_AC_State;

            when goto_RIGHT_SHIFT_AC=>
                  NEXT_STATE <= RIGHT_SHIFT_AC_State;

            when goto_RESET =>
                  Next_State <= AR_gets_PC_State;
            when others =>
      end case;

when DR_gets_AC_State =>
      case Transition is
            when goto_Write_M =>
                  NEXT_STATE <= WRITE_M_State;

            when goto_RESET =>
                  Next_State <= AR_gets_PC_State;

            when others =>
      end case;

when AC_gets_DR_State =>
      case Transition is
            when goto_AR_gets_PC | goto_RESET =>
                  NEXT_STATE <= AR_gets_PC_State;

            when others =>
      end case;

when AR_gets_DR_ADR_State =>
      case Transition is
            when goto_READ_M =>
                  NEXT_STATE <= READ_M_State;

            when goto_DR_gets_AC=>
                  NEXT_STATE <= DR_gets_AC_State;

            when goto_RESET =>
                  Next_State <= AR_gets_PC_State;

            when others =>
      end case;

when AC_gets_AC_plus_DR_State =>
      NEXT_STATE <= AR_gets_PC_State;

when AC_gets_AC_and_DR_State =>
      NEXT_STATE <= AR_gets_PC_State;
```

```
            when AC_gets_NOT_AC_State =>
                  NEXT_STATE <= AR_gets_PC_State;

            when JUMP_State =>
                  NEXT_STATE <= AR_gets_PC_State;

            when WRITE_M_State =>
                  case Transition is
                        when goto_AR_gets_PC =>
                              NEXT_STATE <= AR_gets_PC_State;

                        when others =>
                  end case;

            when RIGHT_SHIFT_AC_State =>
                  NEXT_STATE <= AR_gets_PC_State;

            when Unknown_State =>
                  case Transition is
                        when goto_RESET =>
                              NEXT_STATE <= AR_gets_PC_State;
                        when others =>
                  end case;

            when others =>
      end case;
end process;

-------------------------------------------------------------------
--    The following processes are for the individual states of
--    the State Machine.  These processes handle the output signal
--    assignments and determine the appropriate transition
--    condition in order to exit the state.
-------------------------------------------------------------------

-- AR_gets_PC state
AR_gets_PC: block (Present_State = AR_gets_PC_State) begin
      process (GUARD) begin
            if GUARD then
                  Control <= C10;
                  Transition <= goto_Read_Instruction;
            else
                  Transition <= Null;
                  Control <= null;
            end if;
      end process;
      end block AR_gets_PC;


-- READ_M state
READ_M: block (Present_State = READ_M_State) begin
     process (GUARD) begin
      if GUARD then
            Control <= C3;
      else
            Control <= null;
      end if;
```

```
      end process;

      process (Clock)
       variable clock_count : integer := 0;
      begin
       if GUARD and negedge( clock ) then
            clock_count := clock_count + 1;
            if clock_count = 2 then
                  clock_count := 0;
                  case INSTRUCTION is
                       when LOAD =>
                          Transition <= goto_AC_gets_DR;

                     when ADD =>
                          Transition <= goto_AC_gets_AC_plus_DR;

                     when BIT_AND =>
                          Transition <= goto_AC_gets_AC_and_DR;
                     when others =>
                  end case;
            end if;
       else
            Transition <= Null;
       end if;
      end process;
      end block READ_M;



-- READ_INSTRUCTION state
--    This state not only reads the instruction from memory,
--    but also increments the PC.
READ_Instruction: block (Present_State = READ_Instruction_State) begin
      process (GUARD) begin
            if GUARD then
                  Control <= C9_C3;
            else
                  Control <= null;
            end if;
      end process;

      process (Clock)
            variable clock_count : integer := 0;
      begin
            if GUARD and negedge( clock ) then
                  clock_count := clock_count + 1;
                  if clock_count = 2 then
                        clock_count := 0;
                        Transition <= goto_IR_gets_DR_OP;
                  end if;
            else
                  Transition <= Null;
            end if;
      end process;

      end block READ_Instruction;
```

```
-- IR_gets_DR_OP_State state
IR_gets_DR_OP: block (Present_State = IR_gets_DR_OP_State) begin
      process (GUARD) begin
            if GUARD then
                  control <= C11;

                  case INSTRUCTION is
                  when LOAD | STORE | ADD | BIT_AND =>
                        Transition <= goto_AR_gets_DR_ADR;

                  when JUMPZ =>
                        if ZERO_FLAG ='0' then
                              Transition <= goto_JUMP;
                        else
                              Transition <= goto_AR_gets_PC;
                        end if;

                  when JUMP =>
                        Transition <= goto_JUMP;

                  when COMP =>
                        Transition <= goto_AC_gets_NOT_AC;

                  when RSHIFT =>
                        Transition <= goto_RIGHT_SHIFT_AC;
                  when others =>
                  end case;
            else
                  Transition <= Null;
                  Control <= null;
            end if;
      end process;
      end block IR_gets_DR_OP;

-- DR_gets_AC state
DR_gets_AC: block (Present_State = DR_gets_AC_State) begin
      process (GUARD) begin
            if GUARD then
                  Control <= C5;
                  Transition <= goto_WRITE_M;
            else
                  Transition <= Null;
                  Control <= null;
            end if;
      end process;
      end block DR_gets_AC;

-- AC_gets_DR state
AC_gets_DR: block (Present_State = AC_gets_DR_State) begin
      process (GUARD) begin
            if GUARD then
                  Control <= C6;
                  Transition <= goto_AR_gets_PC;
            else
                  Transition <= Null;
                  Control <= null;
```

```
            end if;
      end process;
      end block AC_gets_DR;

-- AR_gets_DR_ADR state
AR_gets_DR_ADR: block (Present_State = AR_gets_DR_ADR_State) begin
      process (GUARD) begin
            if GUARD then
                  Control <= C7;
                  case INSTRUCTION is
                        when LOAD | BIT_AND | ADD =>
                              Transition <= goto_READ_M;

                        when STORE =>
                              Transition <= goto_DR_gets_AC;

                        when others =>
                  end case,
            else
                  Transition <= Null;
                  Control <= null;
            end if;
      end process;
      end block AR_gets_DR_ADR;

-- AC_gets_AC_plus_DR state
AC_gets_AC_plus_DR: block (Present_State = AC_gets_AC_plus_DR_State)
begin
      process (GUARD) begin
            if GUARD then
                  Control <= C0;
                  Transition <= goto_AR_gets_PC;
            else
                  Transition <= Null;
                  Control <= null;

            end if;
      end process;
      end block AC_gets_AC_plus_DR;

-- AC_gets_AC_and_DR state
AC_gets_AC_and_DR: block (Present_State = AC_gets_AC_and_DR_State) begin
      process (GUARD) begin
            if GUARD then
                  Control <= C1;
                  Transition <= goto_AR_gets_PC;
            else
                  Transition <= Null;
                  Control <= null;
            end if;
      end process;
      end block AC_gets_AC_and_DR;

-- AC_gets_NOT_AC state
AC_gets_NOT_AC: block (Present_State = AC_gets_NOT_AC_State) begin
      process (GUARD) begin
            if GUARD then
```

```
                Control <= C2;
                Transition <= goto_AR_gets_PC;
          else
                Transition <= Null;
                Control <= null;
          end if;
    end process;
    end block AC_gets_NOT_AC;


-- JUMP and JUMPZ state
PC_gets_DR_ADR: block (Present_State = JUMP_State) begin
    process (GUARD) begin
          if GUARD then
                Control <= C8;
                Transition <= goto_AR_gets_PC;

          else
                Transition <= Null;
                Control <= null;
          end if;
    end process;
    end block PC_gets_DR_ADR;


-- WRITE_M state
WRITE_M: block (Present_State = WRITE_M_State) begin
    process ( GUARD ) begin
          if GUARD then
                Control <= C4;
          else
                Control <= null;
          end if;
    end process;

    process (clock)
          variable clock_count : integer := 0;
    begin
          if GUARD and negedge( clock ) then
                clock_count := clock_count + 1;
                if clock_count = 2 then
                      clock_count := 0;
                      Transition <= goto_AR_gets_PC;
                end i ;
          else
                Transition <= null;
          end if;
    end process;
    end block WRITE_M;


-- RIGHT_SHIFT_AC state
RIGHT_SHIFT_AC: block (Present_State = RIGHT_SHIFT_AC_State) begin
    process (GUARD) begin
          if GUARD then
                Control <= C12;
                Transition <= goto_AR_gets_PC;
```

```
            else
                  Transition <= Null;
                  Control <= null;
            end if;
      end process;
      end block RIGHT_SHIFT_AC ;


end BEHAVIORAL;
```

```
------------------------------------------------------------------
--
-- File name:      testbench.vhd
--
-- Description:    testbench for 8 instruction cpu
--                 controller
--
-- Status:  Complete.
--
--
-- Support files: BASICDEFS.vhd      (EIA's BASICDEF's package)
--                 cpu.vhd
--
-- Creation Date: 1 August 90
--
-- Created by:     Rick Miller
--    Address:     AFIT/ENG
--                 Wright-Patterson AFB, OH, 45433
--
--    Phone:       (513)-258-1024  or (513)-255-4960
--
------------------------------------------------------------------

use work.BASICDEFS.all;
use work.CPU_package.all;
use STD.SIMULATOR_STANDARD.all;
use STD.TEXTIO.all;


entity TEST_BENCH is
end TEST_BENCH;


architecture CPU_588 of TEST_BENCH is

  component  CPU_CONTROLLER
      -- generic ();
      port( instruction : in instructions := NOP ;
            CLOCK       : in logic_mv;
            RESET       : in logic_mv     ;
            ZERO_FLAG   : in logic_mv;
            Control_bus :  out logic_vector_mv_bus (12 downto 0)
                        := "XXXXXXXXXXXXX"
            );
    end component;

  for all : CPU_CONTROLLER use entity WORK.CPU_CONTROLLER(BEHAVIORAL);

      signal instruction      : instructions    := NOP;
      signal RESET            : logic_mv  := 'X';
      signal CLOCK            : logic_mv  := 'X';
      signal zero_flag        : logic_mv  := 'X';
      signal control_line     : logic_vector_mv_bus (12 downto 0)
                              := "XXXXXXXXXXXXX";


begin
```

```
process
    file CPU_INSTRUCTIONS  : TEXT is in "CPU_INSTRUCTIONS";
    variable L     : Line;
    variable machine_code : bit_vector(4 downto 0);
      alias machine_instruction : bit_vector(2 downto 0)
                          is machine_code(2 downto 0);
    begin

     readline(CPU_INSTRUCTIONS, L);
     if ENDFILE(CPU_INSTRUCTIONS) then terminate; end if;
     read(L, machine_code);

      case machine_code(4) is
            when '0' => ZERO_FLAG <= '0';
            when '1' => ZERO_FLAG <= '1';
      end case;

      case machine_code(3) is
            when '0' =>  RESET <= '0';
            when '1' =>  RESET <= '1';
      end case;

      wait until control_line = C9_C3;  -- This indicates that the
                                -- CPU_Controller has issued
                                -- a read memory command
                                -- during the READ_INSTRUCTION
                                -- state.

      case machine_instruction is
            when "000" => instruction <= LOAD;
            when "001" => instruction <= STORE;
            when "010" => instruction <= ADD;
            when "011" => instruction <= BIT_AND;
            when "100" => instruction <= JUMP;
            when "101" => instruction <= JUMPZ;
            when "110" => instruction <= COMP;
            when "111" => instruction <= RSHIFT;
      end case;
    end process;

process
    begin
      set_maximums(10000,100);
      tracing_on;
      wait for 1000ns;
      terminate;
    end process;


make_Clock : process
  begin
    wait for 2ns;
    CLOCK <= '0';
    wait for 4ns;
    CLOCK <= '1';
    wait for 2ns;
```

```
  end process make_Clock;

UUT : CPU_CONTROLLER
      port map (  instruction,
                  CLOCK,
                  RESET,
                  ZERO_FLAG,
                  Control_line );


end CPU_588;
```

Vhdl Simulation Report

Report Name: CPU_588"
Kernel Library Name: <<RMILLER.CPU_588>>CPU_588
Kernel Creation Date: AUG-06-1990
Kernel Creation Time: 11:28:25
Run Identifer: 1
Run Date: AUG-06-1990
Run Time: 11:28:25

Report Control Language File: CPU.rcl
Report Output File : cpu_588.rpt

Max Time: 9223372036854775807
Max Delta: 2147483646

Report Control Language :

```
Simulation_report CPU_588 is
begin
report_name is "CPU_588";
page_width is 100;
page_length is 40;
signal_format is horizontal;

sample_signals by_transaction in ns;
--sample_signals by_event in ns;
select_signal : Clock;
select_signal : instruction;
select_signal /uut: reset;
select_signal /uut: zero_flag;
--select_signal /uut: transition;
select_signal : Control_line;
select_signal /UUT: Present_State;
end CPU_588;
```

Report Format Information :

Time is in NS relative to the start of simulation
Time period for report is from 0 NS to End of Simulation
Signal values are reported by transaction (' ' indicates no transaction)

TIME	SIGNAL NAMES					
(NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
0	'X'	NOP	'X'	'X'	"XXXXXXXXXXXXX"	UNKNOWN_STATE
+1			'0'	'0'	"XXXXXXXXXXXXX"	
+2					"ZZZZZZZZZZZZZ"	
1			'1'			
2						
+1	'0'					
4			'0'			
6						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
10						
+1	'0'					
14						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		LOAD				
18						
+1	'0'					
22						
+1	'1'					
+2						READ_INSTRUCTION_STATE
26						
+1	'0'					
30						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
34						
+1	'0'					
38						

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+1	'1'					
+2						AR_GETS_DR_ADR_STATE
+4					"0000010000000"	
42						
+1	'0'					
46						
+1	'1'					
+2						READ_M_STATE
+4					"0000000001000"	
50						
+1	'0'					
54						
+1	'1'					
+2						READ_M_STATE
58						
+1	'0'					
62						
+1	'1'					
+2						AC_GETS_DR_STATE
+4					"0000001000000"	
66						
+1	'0'					
70						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
74						
+1	'0'					
78						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+5		LOAD				
82						
+1	'0'					
86						
+1	'1'					
+2						READ_INSTRUCTION_STATE
90						
+1	'0'					
94						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
98						
+1	'0'					
102						
+1	'1'					
+2						AR_GETS_DR_ADR_STATE
+4					"0000010000000"	
106						
+1	'0'					
110						
+1	'1'					
+2						READ_M_STATE
+4					"0000000001000"	
114						
+1	'0'					
118						
+1	'1'					
+2						READ_M_STATE
122						
+1	'0'					
126						

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+1	'1'					
+2						AC_GETS_DR_STATE
+4					"0000001000000"	
130						
+1	'0'					
134						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
138						
+1	'0'					
142						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		STORE				
146						
+1	'0'					
150						
+1	'1'					
+2						READ_INSTRUCTION_STATE
154						
+1	'0'					
158						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
162						
+1	'0'					
166						
+1	'1'					
+2						AR_GETS_DR_ADR_STATE

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+4					"0000010000000"	
170						
+1			'0'			
174						
+1			'1'			
+2						DR_GETS_AC_STATE
+4					"0000000100000"	
178						
+1			'0'			
182						
+1			'1'			
+2						WRITE_M_STATE
+4					"0000000010000"	
186						
+1			'0'			
190						
+1			'1'			
+2						WRITE_M_STATE
194						
+1			'0'			
198						
+1			'1'			
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
202						
+1			'0'			
206						
+1			'1'			
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		ADD				
210						

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+1	'0'					
214						
+1	'1'					
+2						READ_INSTRUCTION_STATE
218						
+1	'0'					
222						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
226						
+1	'0'					
230						
+1	'1'					
+2						AR_GETS_DR_ADR_STATE
+4					"0000010000000"	
234						
+1	'0'					
238						
+1	'1'					
+2						READ_M_STATE
+4					"0000000001000"	
242						
+1	'0'					
246						
+1	'1'					
+2						READ_M_STATE
250						
+1	'0'					
254						
+1	'1'					
+2						AC_GETS_AC_PLUS_DR_STATE

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+4					"0000000000001"	
258						
+1	'0'					
262						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
266						
+1	'0'					
270						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		BIT_AND				
274						
+1	'0'					
278						
+1	'1'					
+2						READ_INSTRUCTION_STATE
282						
+1	'0'					
286						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
290						
+1	'0'					
294						
+1	'1'					
+2						AR_GETS_DR_ADR_STATE
+4					"0000010000000"	
298						

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+1	'0'					
302						
+1	'1'					
+2						READ_M_STATE
+4					"0000000001000"	
306						
+1	'0'					
310						
+1	'1'					
+2						READ_M_STATE
314						
+1	'0'					
318						
+1	'1'					
+2						AC_GETS_AC_AND_DR_STATE
+4					"0000000000010"	
322						
+1	'0'					
326						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
330						
+1	'0'					
334						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		JUMP				
338						
+1	'0'					
342						

TIME	SIGNAL NAMES					
(NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+1	'1'					
+2						READ_INSTRUCTION_STATE
346						
+1	'0'					
350						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
354						
+1	'0'					
358						
+1	'1'					
+2						JUMP_STATE
+4					"0000100000000"	
362						
+1	'0'					
366						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
370						
+1	'0'					
374						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		JUMPZ				
378						
+1	'0'					
382						
+1	'1'					
+2						READ_INSTRUCTION_STATE

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
386						
+1	'0'					
390						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
394						
+1	'0'					
398						
+1	'1'					
+2						JUMP_STATE
+4					"0000100000000"	
402						
+1	'0'					
406						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
410						
+1	'0'					
414						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		COMP				
418						
+1	'0'					
422						
+1	'1'					
+2						READ_INSTRUCTION_STATE
426						
+1	'0'					

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
430						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
434						
+1	'0'					
438						
+1	'1'					
+2						AC_GETS_NOT_AC_STATE
+4					"0000000000100"	
442						
+1	'0'					
446						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
450						
+1	'0'					
454						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		RSHIFT				
458						
+1	'0'					
462						
+1	'1'					
+2						READ_INSTRUCTION_STATE
466						
+1	'0'					
470						
+1	'1'					

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
474						
+1				'0'		
478						
+1				'1'		
+2						RIGHT_SHIFT_AC_STATE
+4					"1000000000000"	
482						
+1				'0'		
486						
+1				'1'		
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
490						
+1				'0'		
494						
+1				'1'		
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		LOAD				
498						
+1				'0'		
502						
+1				'1'		
+2						READ_INSTRUCTION_STATE
506						
+1				'0'		
510						
+1				'1'		
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	

TIME	SIGNAL NAMES			
(NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
514						
+1	'0'					
518						
+1	'1'					
+2						AR_GETS_DR_ADR_STATE
+4					"0000010000000"	
522						
+1	'0'					
526						
+1	'1'					
+2						READ_M_STATE
+4					"0000000001000"	
530						
+1	'0'					
534						
+1	'1'					
+2						READ_M_STATE
538						
+1	'0'					
542						
+1	'1'					
+2						AC_GETS_DR_STATE
+4					"0000001000000"	
546						
+1	'0'					
550						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
554						
+1	'0'					
558						

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		STORE				
562						
+1	'0'					
566						
+1	'1'					
+2						READ_INSTRUCTION_STATE
570						
+1	'0'					
574						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
578						
+1	'0'					
582						
+1	'1'					
+2						AR_GETS_DR_ADR_STATE
+4					"0000010000000"	
586						
+1	'0'					
590						
+1	'1'					
+2						DR_GETS_AC_STATE
+4					"0000000100000"	
594						
+1	'0'					
598						
+1	'1'					
+2						WRITE_M_STATE

TIME	SIGNAL NAMES					
(NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+4					"0000000010000"	
602						
+1	'0'					
606						
+1	'1'					
+2						WRITE_M_STATE
610						
+1	'0'					
614						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
618						
+1	'0'					
622						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		ADD				
626						
+1	'0'					
630						
+1	'1'					
+2						READ_INSTRUCTION_STATE
634						
+1	'0'					
638						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
642						
+1	'0'					

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
646						
+1	'1'					
+2						AR_GETS_DR_ADR_STATE
+4					"0000010000000"	
650						
+1	'0'					
654						
+1	'1'					
+2						READ_M_STATE
+4					"0000000001000"	
658						
+1	'0'					
662						
+1	'1'					
+2						READ_M_STATE
666						
+1	'0'					
670						
+1	'1'					
+2						AC_GETS_AC_PLUS_DR_STATE
+4					"0000000000001"	
674						
+1	'0'					
678						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
682						
+1	'0'					
686						
+1	'1'					
+2						READ_INSTRUCTION_STATE

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+4					"0001000001000"	
+5		BIT_AND				
690						
+1	'0'					
694						
+1	'1'					
+2						READ_INSTRUCTION_STATE
698						
+1	'0'					
702						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
706						
+1	'0'					
710						
+1	'1'					
+2						AR_GETS_DR_ADR_STATE
+4					"0000010000000"	
714						
+1	'0'					
718						
+1	'1'					
+2						READ_M_STATE
+4					"0000000001000"	
720				'1'		
722						
+1	'0'					
726						
+1	'1'					
+2						READ_M_STATE
730						

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+1	'0'					
734						
+1	'1'					
+2						AC_GETS_AC_AND_DR_STATE
+4					"0000000000010"	
738						
+1	'0'					
742						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
746						
+1	'0'					
750						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		JUMP				
754						
+1	'0'					
758						
+1	'1'					
+2						READ_INSTRUCTION_STATE
762						
+1	'0'					
766						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
770						
+1	'0'					
774						

TIME (NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+1	'1'					
+2						JUMP_STATE
+4					"0000100000000"	
778						
+1	'0'					
782						
+1	'1'					
+2						AR_GETS_PC_STATE
+4					"0010000000000"	
786						
+1	'0'					
790						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	
+5		JUMPZ				
794						
+1	'0'					
798						
+1	'1'					
+2						READ_INSTRUCTION_STATE
802						
+1	'0'					
806						
+1	'1'					
+2						IR_GETS_DR_OP_STATE
+4					"0100000000000"	
810						
+1	'0'					
814						
+1	'1'					
+2						AR_GETS_PC_STATE

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION	RESET	ZERO_FLAG	CONTROL_LINE(12 DOWNTO 0)	PRESENT_STATE
+4					"0010000000000"	
818						
+1	'0'					
822						
+1	'1'					
+2						READ_INSTRUCTION_STATE
+4					"0001000001000"	

Appendix C: The Structural Model.

This appendix contains a complete listing of all logic gates and flip-flops defined for use within a sequential circuit's architecture. Additionally, another example structural architecture of a sequence detector is included. The logic gates and flip-flops are:

1. ANDm — A multiple input AND gate.
2. NANDm — A multiple input NAND gate.
3. ORm — A multiple input OR gate.
4. NORm — A multiple input NOR gate.
5. INVERTER — a single input, single output inverter.
6. D_ff — A clocked D flip-flop with set and clear, and,
7. D_ff_no_clk — An asynchronous D flip-flop with set and clear.

They may be found on the following pages:

```
------------------------------------------------------------------
--
--          ANDm and NANDm logic gates
--
------------------------------------------------------------------

use work.BASICDEFS.all;
entity ANDm is
      generic(
            propagation_delay : time := 0 ns
            );
      port ( In1   : in logic_vector_mv ;
            out1   : out logic_mv := 'U'
            );
end ANDm;

architecture BEHAVIORAL of ANDm is
begin

      out1 <= and_bw( In1 ) after propagation_delay;

end BEHAVIORAL;



use work.BASICDEFS.all;
entity NANDm is

      generic(
            propagation_delay : time := 0 ns
            );

      port(In1     : in logic_vector_mv;
            out1   : out logic_mv := 'U'
            );

end NANDm;

architecture BEHAVIORAL of NANDm is
begin

      out1 <= nand_bw( In1 ) after propagation_delay;

end BEHAVIORAL;
```

```
------------------------------------------------------------------
--
--          ORm and NORm logic gates
--
------------------------------------------------------------------

use work.BASICDEFS.all;
entity ORm is
      generic(
            propagation_delay : time := 0 ns
            );
      port(In1    : in logic_vector_mv ;
            out1  : out logic_mv := 'U'
            );
end ORm;

architecture BEHAVIORAL of ORm is
begin

      out1 <=  or_bw( In1 ) after propagation_delay;

end BEHAVIORAL;



use work.BASICDEFS.all;
entity NORm is
      generic(
            propagation_delay : time := 0 ns
            );
      port(In1    : in logic_vector_mv ;
            out1  : out logic_mv := 'U'
            );
end NORm;

architecture BEHAVIORAL of NORm is
begin

      out1 <= nor_bw( In1 ) after propagation_delay;

end BEHAVIORAL;
```

```
------------------------------------------------------------------
--
--          Inverter logic gate
--
------------------------------------------------------------------

use work.BASICDEFS.all;
entity inverter is
      generic(
            propagation_delay : time := 0 ns
            );
      port (
            in1   : in logic_mv := 'U';
            out1  : out logic_mv := 'U'
            );
end inverter;

architecture behavioral of inverter is
begin

      out1 <= not in1 after propagation_delay;

end behavioral;
```

```
-------------------------------------------------------------------
--
--          D flip-flop with clock, set, and clear
--
-------------------------------------------------------------------

use work.BASICDEFS.all;
entity D_ff is
      generic ( propagation_delay     : time     := 0 ns );
      port (
         D_in      : in logic_mv       := 'U';
         Q_out     : out logic_mv      := 'U';
         CLK       : in logic_mv       := 'U';
         Clear     : in logic_mv       := 'U';
         SET       : in logic_mv       := 'U'
         );
end D_ff;

architecture BEHAVIORAL of D_ff is

begin
      process (CLK)
      begin
         if (CLK'event and CLK = '1') then

            if ((Clear = '1') and (SET = '1')) then
                  Q_out <= 'U' after propagation_delay; end if;

            if ((Clear = '0') and (SET = '1')) then
                  Q_out <= '1' after propagation_delay ; end if;

            if ((Clear = '1') and (SET = '0')) then
                  Q_out <= '0' after propagation_delay; end if;

            if ((Clear = '0') and (SET = '0')) then
                  Q_out <= D_in after propagation_delay; end if;

         end if;
      end process;

end BEHAVIORAL;
```

```
------------------------------------------------------------------
--
--          Asynchronous D flip-flop with set and clear
--
------------------------------------------------------------------

use work.BASICDEFS.all;
entity D_ff_no_clk is
      generic ( propagation_delay   : time      := 0 ns );
      port (
            D_in  : in logic_mv     := 'U';
            Q_out : out logic_mv    := 'U';
            Clear : in logic_mv     := 'U';
            SET   : in logic_mv     := 'U'
            );
end D_ff_no_clk;


architecture BEHAVIORAL of D_ff_no_clk is

begin
      process ( Clear, SET, D_in )
      begin

            if ((Clear = '1') and (SET = '1')) then

                  Q_out <= 'U' after propagation_delay; end if;

            if ((Clear = '0') and (SET = '1')) then
                  Q_out <= '1' after propagation_delay ; end if;

            if ((Clear = '1') and (SET = '0')) then
                  Q_out <= '0' after propagation_delay; end if;

            if ((Clear = '0') and (SET = '0')) then
                  Q_out <= D_in after propagation_delay; end if;

      end process;

end BEHAVIORAL;
```

```
------------------------------------------------------------------
--
--          Sequence Detector constructed of NAND logic
--
------------------------------------------------------------------

use WORK.basicdefs.all;

entity Sequence_Detector is
      -- generic ( );

      port (
            Xin   : in logic_mv     := 'U';
            CLOCK : in logic_mv     := 'U';
            Zout  : out logic_mv    := 'U';
            SET   : in logic_mv     := 'U';
            CLEAR : in logic_mv     := 'U'
           );
end Sequence_Detector;


architecture STRUCTURAL2 of Sequence_Detector is

 component inverter
      generic( propagation_delay : time := 0 ns );
      port (
            in1   : in logic_mv     := 'U';
            out1  : out logic_mv    := 'U'
            );
  end component;

      for all : inverter use
                  entity WORK.inverter(BEHAVIORAL);

  component NANDm
        generic( propagation_delay : time := 0 ns );
        port(In1  : in logic_vector_mv;
                out1    : out logic_mv := 'U'
                );
  end component;

      for all : NANDm use
                  entity WORK.NANDm(BEHAVIORAL);

  component D_ff
      generic ( propagation_delay   : time := 0 ns );
      port (
            D_in        : in logic_mv     := 'U';
            Q_out       : out logic_mv    := 'U';
            CLK         : in logic_mv     := 'U';
            Clear : in logic_mv     := 'U';
            SET         : in logic_mv     := 'U'
            );
  end component;

      for all : d_ff use
                  entity WORK.D_ff(BEHAVIORAL);
```

```
  --- Internal Signal Declarations

      signal  Y1,
              Y2,
              Q1,
              Q2 : logic_mv := 'U';

      signal  Xnot,
              Q1not,
              Q2not,
              NAND1_output,
              NAND2_output,
              NAND3_output,
              NAND4_output : logic_mv := 'U';

      signal  NAND1_input,
              NAND2_input,
              NAND3_input,
              OR1_input : logic_vector_mv (1 downto 0) := "UU";

      signal  NAND4_input : logic_vector_mv (2 downto 0) := "UUU";

begin

  inv1 : inverter
           port map (Xin, Xnot);

  inv2 : inverter
           port map (Q2, Q2not);

  inv3 : inverter
           port map (Q1, Q1not);

  ---  Logic to derive Y1:

      NAND1_input <= Q1 & Q2not;

      NAND1 : NANDm
            port map ( NAND1_input, NAND1_output );

      NAND2_input <= Xnot & NAND1_output;

      NAND2 : NANDm
            port map ( NAND2_input, Y1 );

  --- LOGIC to derive Y2

      NAND3_input <= Xnot & Q1;

      NAND3 : NANDm
            port map ( NAND3_input, NAND3_output );

      inv4 : inverter
            port map ( NAND3_output, Y2 );

  --- LOGIC to derive Zout
```

```
        NAND4_input <= Xin & Q1not & Q2;

        NAND4 : NANDm
              port map ( NAND4_input, NAND4_output );

        inv5  : inverter
              port map ( NAND4_output , Zout );

    --- Registers

    FF1 : D_ff
              port map ( Y1, Q1, CLOCK, CLEAR, SET );

    FF2 : D_ff
              port map ( Y2, Q2, CLOCK, CLEAR, SET  );


end STRUCTURAL2;
```

Vhdl Simulation Report

Report Name: TEST_GATE"
Kernel Library Name: <<RMILLER.STRUCTURAL>>SD_TEST
Kernel Creation Date: AUG-01-1990
Kernel Creation Time: 10:09:35
Run Identifer: 1
Run Date: AUG-01-1990
Run Time: 10:09:35

Report Control Language File: test.rcl
Report Output File : sd_test.rpt

Max Time: 9223372036854775807
Max Delta: 2147483646

Report Control Language :

```
Simulation_report TEST is
begin
report_name is "TEST_GATE";
page_width is 80;
page_length is 50;
signal_format is horizontal;

sample_signals by_transaction in ns;
--sample_signals by_event in ns;
select_signal : Clock;
--select_signal : reset;
select_signal : instruction;
select_signal : out_1;
select_signal /uut1: Y2;
select_signal /uut1: Y1;
select_signal /uut1: Q2;
select_signal /uut1: Q1;

end TEST;
```

Report Format Information :

Time is in NS relative to the start of simulation
Time period for report is from 0 NS to End of Simulation
Signal values are reported by transaction (' ' indicates no transaction)

TIME	SIGNAL NAMES						
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	OUT_1	Y2	Y1	Q2	Q1
0	'U'	"UUU"	'U'	'U'	'U'	'U'	'U'
+1		"100"	'X'	'X'	'X'		
+2			'X'	'X'			
+3					'X'		
+4			'0'		'X'		
2							
+1	'1'						
+2						'0'	'0'
+5				'0'			
+6					'0'		
6							
+1	'0'						
8							
+1		"000"					
10							
+1	'1'						
+2						'0'	'0'
14							
+1	'0'						
16							
+1		"000"					
18							
+1	'1'						
+2						'0'	'0'
22							
+1	'0'						
24							
+1		"001"					
+4					'1'		
26							
+1	'1'						
+2						'0'	'1'
+6					'1'		
30							
+1	'0'						
32							
+1		"000"					
+4					'1'		
+5				'1'			
34							
+1	'1'						
+2						'1'	'1'

TIME	SIGNAL NAMES						
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	OUT_1	Y2	Y1	Q2	Q1
+7					'0'		
38							
+1	'0'						
40							
+1		"000"					
42							
+1	'1'						
+2						'1'	'0'
+5				'0'			
46							
+1	'0'						
48							
+1		"001"					
+4			'1'		'1'		
50							
+1	'1'						
+2						'0'	'1'
+5			'0'				
+7					'1'		
54							
+1	'0'						
56							
+1		"000"					
+4					'1'		
+5				'1'			
58							
+1	'1'						
+2						'1'	'1'
+7					'0'		
62							
+1	'0'						
64							
+1		"001"					
+4					'1'		
+5				'0'			
66							
+1	'1'						
+2						'0'	'1'
+7					'1'		
70							
+1	'0'						
72							

TIME	SIGNAL NAMES							
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	OUT_1	Y2	Y1	Q2	Q1
+1		"001"					
74							
+1	'1'						
+2						'0'	'1'
78							
+1	'0'						
80							
+1		"001"					
82							
+1	'1'						
+2						'0'	'1'
86							
+1	'0'						
88							
+1		"000"					
+4					'1'		
+5				'1'			
90							
+1	'1'						
+2						'1'	'1'
+7					'0'		
94							
+1	'0'						
96							
+1		"000"					
98							
+1	'1'						
+2						'1'	'0'
+5				'0'			
102							
+1	'0'						
104							
+1		"011'					
+4			'1'		'1'		
106							
+1	'1'						
+2						'1'	'1'
+6			'0'				
110							
+1	'0'						

Appendix D: The Verification Software Environment

This appendix serves as a user's manual for AFIT's verification software environment which consists of UC Berkeley's pre_verif and verif software and AFIT's b2s software. Although it is intended as a standalone document, additional information which amplifies the contents of this appendix may be found in Chapters 2, 3, 4, and 5 of the thesis. Where applicable, references will be provided in this appendix to appropriate chapters of the thesis.

D.1 The Verification Software Environment.

Figure D.1 represents AFIT's verification software environment which performs verification of sequential circuits which have been described in the behavioral or structural models defined in Chapter 3 of this thesis. Verification may be performed between two structurally described circuits, two behaviorally described circuits, or one of each. Chapter 4 describes the input file format for both b2s and pre_verif. The intermediary file, verif.input, is described in Chapter 2. All three software tools, pre_verif, verif, and b2s may be found on AFIT's VLSI sun network in the directory

```
/tmp_mnt/auto/project/verification
```

The b2s software is currently available in source and executable form on the AFIT VLSI suns. The software pre_verif and verif are available in source on the AFIT VLSI suns; currently they are executable only on a microvax.

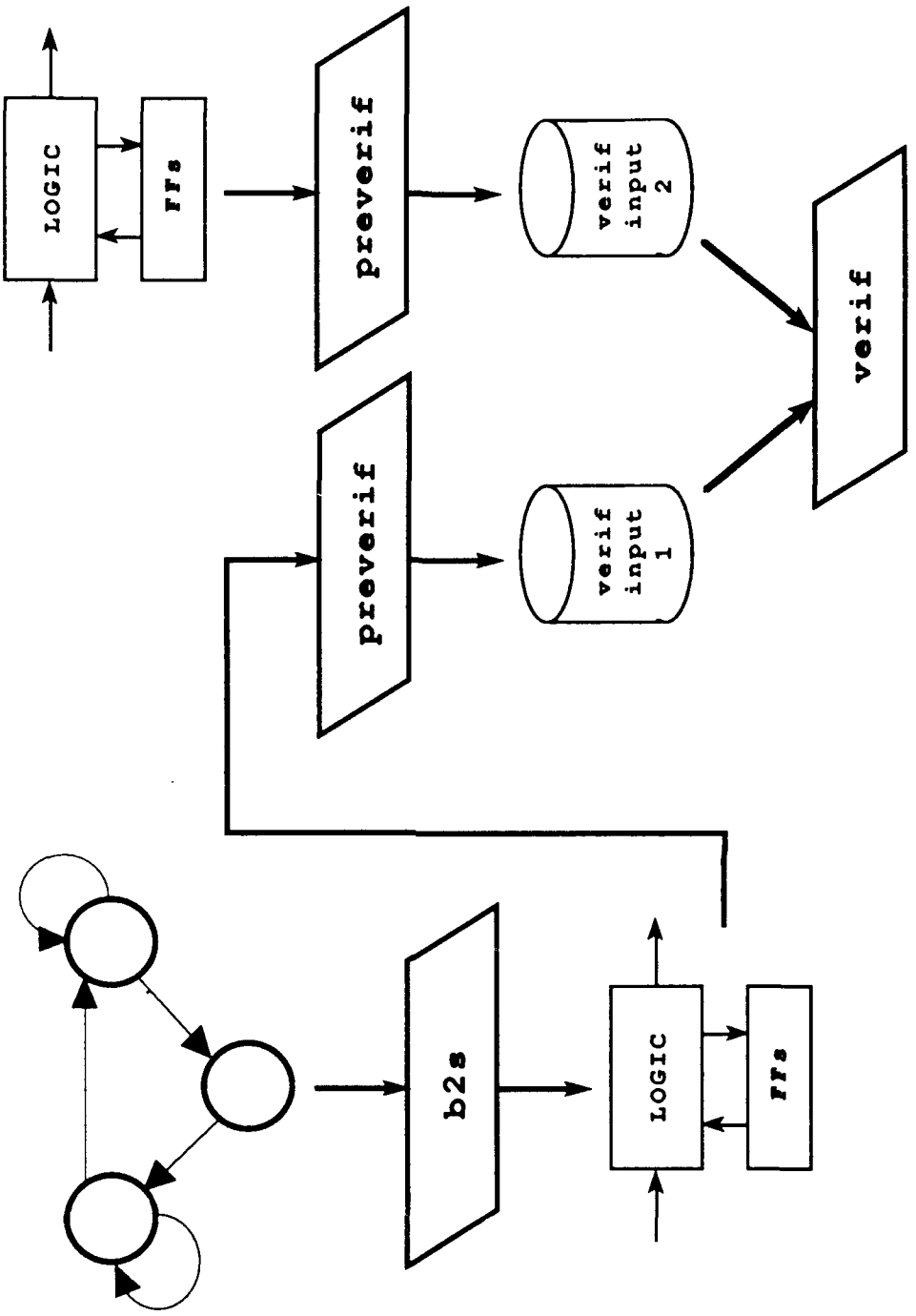


Figure D.1 The Verification Software Environment.

D.2 Software Tutorial.

For the purposes of demonstration, these instructions will step through the verification of one behaviorally described circuit against a structurally defined circuit. These circuits are labeled with the numbers 1 and 2 in Figure D.1. First, the behaviorally specified sequential circuit will be processed through b2s and pre_verif. Next, the structurally modeled circuit will be processed through pre_verif. Each pre_verif run will produce an input file for the verif software. Finally, verif will determine the equivalence of the two verif.input files. Both VHDL files can be found at the end of this appendix.

D.2.1 The b2s Software.

The b2s software translates a behaviorally specified sequential circuit into a structurally equivalent circuit. The input behavioral specification must be in the behavioral VHDL format described in Chapter 4. Currently, b2s accepts VHDL behavioral models of only simple synchronous or asynchronous Mealy sequential circuits. Further, although b2s does perform some source file checking, the input VHDL source code is assumed to be syntactically and semantically correct. The b2s software produces a structural VHDL output file formatted in the structural VHDL model presented in Chapter 4. The b2s software is invoked from the system prompt by typing:

```
b2s [-options] filename
```

Where `[-options]` allows some user visibility into b2s's translation execution and filename is any legal Unix filename. The file contains the behavioral VHDL model of the sequential circuit. The b2s options permitted for `[-options]` are:

-de which causes b2s to print to stdout information regarding the sequential circuit's VHDL entity.

-da which causes b2s to print to stdout translation information regarding the sequential circuit's architectural body.

-dt which causes b2s to print to stdout translation information regarding the sequential circuit's transition process,

-db which causes b2s to print to stdout translation information regarding the sequential circuit's block constructs,

-dA which causes b2s to print to stdout all translation information, and,

-dT which causes b2s to print the state transition table generated from the behavioral description.

Additionally, multiple options may be invoked by concatenation, for example:

```
b2s -dedT filename
```

causes execution of both the -de and -dT options. The b2s software puts the structurally equivalent VHDL circuit in the file:

```
filename.struc
```

This output file name is created by appending `.struc` to the input file name. This new structural file is represented by the number 3 in Figure D.1.

As a final note, if b2s encounters any difficulties translating the behavioral VHDL circuit, carefully examine the nature of the behavioral file. Currently, b2s limits the "free-form" nature of the VHDL which it accepts. Although the behavioral VHDL is completely analyzable and simulatable by the VHDL software support environment, b2s is not a VHDL source code analyzer! In its current form it expects certain VHDL constructs to be formatted in a certain fashion; Chapter 4 presents these formats for the behavioral VHDL. Additionally, section D.4.1 presents an example.

D.2.2 The pre_verif Software.

The pre_verif software takes a structurally designed sequential circuit and extracts the circuit's complete cover and minterm information. This information is placed in an output file named verif.input. See Chapter 2 for further information regarding the contents of verif.input. The pre_verif software is invoked from the system prompt by typing:

```
pre_verif [-options] filename
```

Where [-options] controls pre_verif's operation and filename is the name of the input file containing the structurally designed circuit Many `[-options]` are possible for the pre_verif software. Simply type:

```
pre_verif
```

at the system prompt for a complete listing. Only two of these options are of interest here, namely `-enum` and `-vhdl`. The first option, `-enum`, instructs pre_verif to extract the complete cover and minterm variable information and place this information in the output file verif.input. The second option, `-vhdl`, informs pre_verif that the input file is a sequential circuit specified using the VHDL structural model of Chapter 3. Lacking the `-vhdl` option, pre_verif will expect the input file to be in the UC Berkeley structural netlist format. The complete command line invocation of pre_verif using a VHDL structural input file is:

```
pre_verif -enum -vhdl filename
```

where filename is the name of the VHDL structural file and can be any legal Unix filename. The two options, `-vhdl` and `-enum`, are interchangeable in that:

```
pre_verif -vhdl -enum filename
```

is equivalent.

For purposes of the demonstration, the pre_verif software is invoked twice, once for the structural file created by the b2s software (labeled number 3 in Figure D.1) and again for the other structural file (labeled number 2 in Figure D.1). The two files created after both runs of pre_verif are labeled with the number 4 in Figure D.1. One caution must be taken after each invocation of the pre_verif software. Because pre_verif always places the cover and minterm variable information into a file named verif.input, the user should change this file's name to prevent subsequent pre_verif executions to delete the old verif.input file and in so doing, lose the information from previous pre_verif runs. The verif.input file name can be easily changed at the Unix system prompt by typing:

```
mv verif.input filename
```

where filename is the new file name designated by the user.

D.2.3 The verif Software.

The verif software performs the actual verification, or equivalence check, of the two sequential circuits. The verif software does not create an output file but simply prints verification information to Unix's stdout. The software is invoked from the system prompt by typing:

```
verif filename1 filename2
```

Where `filename1` and `filename2` are the names of the two files created by the two separate pre_verif runs. If the two sequential circuits are equivalent, verif will exit with the message:

```
#MACHINES ARE THE SAME
```

If the two circuits are not equivalent, verif will exit with the message:

```
#MACHINES ARE DIFFERENT
```

Accompanying this message will be a set of vectors which are the differentiating sequence for the two machines. For example:

```
#MACHINES ARE DIFFERENT
#THE DISTINGUISHING SEQUENCE IS :
--1-
--1-
--1-
--1-
1010
----
```

These vectors describe the input sequence which steps through the sequential circuits starting from the each circuit's initial state. The state reached upon application of the last vector is that state which is different than the second machine's state. The two machines may be debugged at this point by going directly to the state transition graph for each machine and tracing through the paths via the vectors or by applying the vectors directly to the sequential circuit's VHDL simulation.

D.3 Summary of Verification Process

The steps involved to verify the equivalence of two sequential circuits where one circuit is described using the behavioral VHDL model (`filename1`) and the other using the structural VHDL model (`filename2`), can be summarized as follows:

(1) Run b2s to translate the behavioral circuit into a structural equivalent :

```
b2s filename1
```

(2) Run pre_verif on b2s's output file:

```
pre_verif -enum -vhdl filename1.struc
```

(3) Rename the file created by pre_verif:

```
mv verif.input filename1.verif
```

(4) Run pre_verif on the second file:

```
pre_verif -enum -vhdl filename2
```

(5) Perform the verification:

```
verif filename1.verif verif.input
```

(6) Should verif report that the circuits are different, record the vectors which verif reports.

D.4 An Example b2s Translation

The following VHDL code describes the sequential circuit of Figure D.2. Section D.4.1 contains the behavioral VHDL code describing the circuit and Section D.4.2 contains the structural VHDL code generated by the b2s software. This example is available in the directory:

```
/tmp_mnt/auto/project/verification/b2s
```

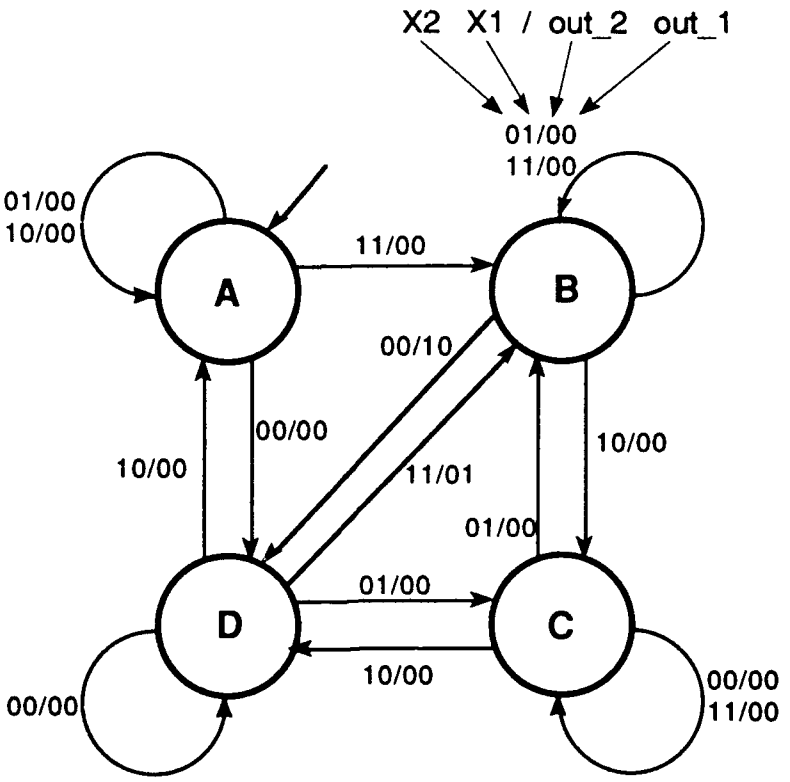


Figure D.2 Example Sequential Circuit.

D.4.1 The VHDL Behavioral Version

```
use work.example_pkg.all;

use work.BASICDEFS.all;


entity example is

      port(
            X1          : in  logic_mv := 'U';
            X2          : in  logic_mv := 'U';
            out_1       : out logic_mv := 'U';
            out_2       : out logic_mv := 'U';
            INITIALIZE  : in  logic_mv;
            CLOCK       : in  logic_mv

            );

end example;
```

```
architecture SYNCHRONOUS of example is


      signal  Present_State, Next_State : States
                                 := Unknown_State;

         signal          Transition :      Transition_Resolution
                               Transition_Conditions
                               BUS := No_Transition;


begin

-- synchronize the state to state transitions to the clock.
-- Note that signals on a process's sensitivity list must be
-- separated from the parenthesis by spaces!
process ( clock )
  begin
      if (clock = '1' and clock'event)
            then Present_State <= Next_State;
      end if;
  end process;

-- initialize and reset capability
-- Note that signals on a process's sensitivity list must be
-- separated from the parenthesis by spaces!
process ( INITIALIZE )
  begin
      if  INITIALIZE = '1' and INITIALIZE'event then
            Transition <= INITIALIZE;
      else
            Transition <= null;
      end if;
  end process;

-- the State Machine Transitions
-- Note that signals on a process's sensitivity list must be
-- separated from the parenthesis by spaces!
process ( transition )
  begin
  case Present_State is

      when State_C =>
            case transition is
                  -- Notice that not all transitions in Figure D.2
                  -- are enumerated here.  The ones not enumerated
                  -- are those which are transitions back into the
                  -- same state.  They do not need to be enumerated;
                  -- in their absence, b2s adds them to the
                  -- State Transition Table.
                  when zero_one =>
                        Next_State <= State_B;
                  when one_zero =>
                        Next_State <= State_D;
                  -- Additionally, b2s reserves two transition names:
                  -- INITIALIZE and RESET.  These two
                  -- transitions must be used to transition the
```

```
                -- circuit into it's initial state.  They are
                -- utilized ONLY IN THE VHDL; b2s ignores them.
                when INITIALIZE =>
                      Next_State <= State_A;
                when others =>
          end case;

    when State_D =>
          case transition is
                when zero_one =>
                      Next_State <= State_C;
                when one_one =>
                      Next_State <= State_B;
                when one_zero =>
                      Next_State <= State_A;
                when INITIALIZE =>
                      Next_State <= State_A;
                when others =>
          end case;

    when State_B =>
          case transition is
                when zero_zero =>
                      Next_State <= State_D;
                when one_zero =>
                      Next_State <= State_C;
                when INITIALIZE =>
                      Next_State <= State_A;
                when others =>
          end case;

    when State_A =>
          case transition is
                when zero_zero =>
                      Next_State <= State_D;
                when one_one =>
                      Next_State <= State_B;
                when INITIALIZE =>
                      Next_State <= State_A;
                when others =>
          end case;


    when Unknown_State =>
          case transition is
                when INITIALIZE =>
                      Next_State <= State_A;
                when others =>
          end case;

    end case;
  end process;


--  C State block
C: block ( Present_State = State_C )
      signal inputs : logic_vector_mv ( 1 downto 0 );
```

```
  begin
      -- the block signal "inputs" need not consist of all
      -- input port signals of the entity.  It should only consist
      -- of those inputs required for the state to operate
      -- properly.  Leaving out extraneous input port signals
      -- reduces the variable count in the generated minterms.
      -- For this example, however, both X2 and X1 are required.
      inputs <= X2 & X1;
      process (GUARD, Inputs )
      begin
      if GUARD then
        case inputs is
            when "01" =>
                  transition <=  zero_one;

            when "10" =>
                  transition <=  one_zero;

            -- As mentioned in the transition process, not all
            -- transitions are enumerated.  If they were, the
            -- following lines of code, which have been commented
            -- out would be required.  Implied their absence, b2s
            -- inserts them into the state transition table.
            --when "00"
            --    transition <= zero_zero;
            --when "11"
            --    transition <= one_one;

            --when others =>
            --    transition <= No_Transition;

        end case;
      else
            transition <= null;
      end if;
      end process;

      end block C;

--  D State block
D: block ( Present_State = State_D )
      signal inputs : logic_vector_mv ( 1 downto 0 );
  begin
      inputs <= X2 & X1;
      process ( GUARD, Inputs )
      begin
      if GUARD then
        case inputs is
            when "10" =>
                  transition <=  one_zero;

            when "11" =>
                  transition <=  one_one;
                  out_1 <= '1';

            when "01" =>
                  transition <=  zero_one;
```

```
            when others =>
                  transition <= No_Transition;

        end case;
      else
            transition <= null;
            out_1 <= null;
      end if;
      end process;

      end block D;

--  B State block
B: block ( Present_State = State_B )
      signal inputs : logic_vector_mv ( 1 downto 0 );
  begin
      inputs <= X2 & X1;
      process (GUARD, Inputs )
      begin
      if GUARD then
        case inputs is
            when "00" =>
                  transition <=  zero_zero;
                  out_2 <= '1';

            when "10" =>
                  transition <=  one_zero;

            when others =>
                  transition <= No_Transition;

        end case;
      else
            transition <= null;
            out_2 <= null;
      end if;
      end process;

      end block B;


--  A State block
A: block ( Present_State = State_A )
      signal inputs : logic_vector_mv ( 1 downto 0 );
  begin
      inputs <= X2 & X1;
  process (GUARD, Inputs )
      begin
      if GUARD then
        case inputs is
            when "00" =>
                  transition <=  zero_zero;

            when "11" =>
                  transition <=  one_one;

            when others =>
                  transition <= No_Transition;
```

```
        end case;
      else
            transition <= null;
      end if;
  end process;

      end block A;

end SYNCHRONOUS;
```

D.4.2 b2s's Structural Translation

The b2s software produces the following structural equivalent of the behavioral description of section D.4.1.

```
-------------------------------------------------
--                                             --
--  Structural VHDL created by the b2s software  --
--                                             --
--  Captain Rick Miller                        --
--  AFIT/ENG                                   --
--                                             --
--  Copyright (c) 9 November 1990.             --
--                                             --
-------------------------------------------------

use work.example_pkg.all;

use work.BASICDEFS.all;


entity example is

      port(
            X1          : in  logic_mv := 'U';
            X2          : in  logic_mv := 'U';
            out_1       : out logic_mv := 'U';
            out_2       : out logic_mv := 'U';
            INITIALIZE  : in  logic_mv;
            CLOCK       : in  logic_mv

            );

end example;


architecture STRUCTURAL of example is

  --- Component Declarations

 component inverter
      generic( propagation_delay : time := 0 ns );
      port (
            in1   : in logic_mv     := 'U';
            out1  : out logic_mv    := 'U'
            );

  end component;

      for all : inverter use
                  entity WORK.inverter(BEHAVIORAL);


 component ANDm
```

```
    generic( propagation_delay : time := 0 ns );
    port (
          In1    : in logic_vector_mv;
          out1   : out logic_v     := 'U'
          );

end component;

    for all : ANDm use
                entity WORK.ANDm(BEHAVIORAL);


component ORm
    generic( propaga ion_delay : time := 0 ns );
    port (
          In1    : in logic_vector_mv;
          out1   : out logic_mv     := 'U'
          );

end component;

    for all : ORm use
                entity WORK.ORm(BEHAVIORAL);


component D_ff
    generic( propagation_delay : time := 0 ns );
    port (
          D_in   : in logic_mv      := 'U';
          Q_out        : out logic_mv     := 'U';
          CLK    : in logic_mv      := 'U';
          Clear  : in logic_mv      := 'U';
          SET    : in logic_mv      := 'U'
          );

end component;

    for all : D_ff use
                entity WORK.D_ff(BEHAVIORAL);


--- Internal Signal Declarations

    signal Qinit : logic_vector_mv ( 1 downto 0 );
    signal out_1_in0 : logic_vector_mv ( 3 downto 0 );
    signal out_2_in0 : logic_vector_mv ( 3 downto 0 );
    signal Y1_in7 : logic_vector_mv ( 3 downto 0 );
    signal Y1_in6 : logic_vector_mv ( 3 downto 0 );
    signal Y1_in5 : logic_vector_mv ( 3 downto 0 );
    signal Y1_in4 : logic_vector_mv ( 3 downto 0 );
    signal Y1_in3 : logic_vector_mv ( 3 downto 0 );
    signal Y1_in2 : logic_vector_mv ( 3 downto 0 );
    signal Y1_in1 : logic_vector_mv ( 3 downto 0 );
    signal Y1_in0 : logic_vector_mv ( 3 downto 0 );
    signal OR_Y1in : logic_vector_mv ( 7 downto 0 );
    signal Y0_in8 : logic_vector_mv ( 3 downto 0 );
    signal Y0_in7 : logic_vector_mv ( 3 downto 0 );
    signal Y0_in6 : logic_vector_mv ( 3 downto 0 );
```

```
        signal Y0_in5 : logic_vector_mv ( 3 downto 0 );
        signal Y0_in4 : logic_vector_mv ( 3 downto 0 );
        signal Y0_in3 : logic_vector_mv ( 3 downto 0 );
        signal Y0_in2 : logic_vector_mv ( 3 downto 0 );
        signal Y0_in1 : logic_vector_mv ( 3 downto 0 );
        signal Y0_in0 : logic_vector_mv ( 3 downto 0 );
        signal OR_Y0in : logic_vector_mv ( 8 downto 0 );
        signal Q0_NOT : logic_mv;
        signal Q0 : logic_mv;
        signal Y0 : logic_mv;
        signal Q1_NOT : logic_mv;
        signal Q1 : logic_mv;
        signal Y1 : logic_mv;
        signal X2_NOT : logic_mv;
        signal X1_NOT : logic_mv;

begin

 g0 : inverter
            port map( X1, X1_NOT );

 g1 : inverter
            port map( X2, X2_NOT );

 g2 : inverter
            port map( Q1, Q1_NOT );

 g3 : inverter
            port map( Q0, Q0_NOT );

      -- Following combinational logic generates flip-flop input(s).

      -- The following combo logic generates: Y0

      Y0_in0 <= X2 & X1 & Q1 & Q0;

 g4 : ANDm
      port map( Y0_in0 , Y0_0 );

      Y0_in1 <= X2_NOT & X1_NOT & Q1 & Q0;

 g5 : ANDm
      port map( Y0_in1 , Y0_1 );

      Y0_in2 <= X2 & X1 & Q1_NOT & Q0;

 g6 : ANDm
      port map( Y0_in2 , Y0_2 );

      Y0_in3 <= X2_NOT & X1 & Q1_NOT & Q0;

 g7 : ANDm
      port map( Y0_in3 , Y0_3 );

      Y0_in4 <= X2_NOT & X1 & Q1 & Q0;

 g8 : ANDm
      port map( Y0_in4 , Y0_4 );
```

```
     Y0_in5 <= X2_NOT & X1 & Q1 & Q0_NOT;

g9 : ANDm
     port map( Y0_in5 , Y0_5 );

     Y0_in6 <= X2 & X1 & Q1 & Q0_NOT;

g10 : ANDm
     port map( Y0_in6 , Y0_6 );

     Y0_in7 <= X2 & X1_NOT & Q1_NOT & Q0;

g11 : ANDm
     port map( Y0_in7 , Y0_7 );

     Y0_in8 <= X2 & X1 & Q1_NOT & Q0_NOT;

g12 : ANDm
     port map( Y0_in8 , Y0_8 );

     OR_Y0in <= Y0_0 & Y0_1 & Y0_2 & Y0_3 & Y0_4 & Y0_5 & Y0_6 & Y0_7 & Y0_8;
g13 : ORm
     port map( OR_Y0in , Y0 );

     -- The following combo logic generates: Y1

     Y1_in0 <= X2 & X1 & Q1 & Q0;

g14 : ANDm
     port map( Y1_in0 , Y1_0 );

     Y1_in1 <= X2_NOT & X1_NOT & Q1 & Q0;

g15 : ANDm
     port map( Y1_in1 , Y1_1 );

     Y1_in2 <= X2_NOT & X1_NOT & Q1 & Q0_NOT;

g16 : ANDm
     port map( Y1_in2 , Y1_2 );

     Y1_in3 <= X2 & X1_NOT & Q1 & Q0;

g17 : ANDm
     port map( Y1_in3 , Y1_3 );

     Y1_in4 <= X2_NOT & X1 & Q1 & Q0_NOT;

g18 : ANDm
     port map( Y1_in4 , Y1_4 );

     Y1_in5 <= X2_NOT & X1_NOT & Q1_NOT & Q0;

g19 : ANDm
     port map( Y1_in5 , Y1_5 );

     Y1_in6 <= X2 & X1_NOT & Q1_NOT & Q0,
```

```
g20 : ANDm
     port map( Y1_in6 , Y1_6 );

     Y1_in7 <= X2_NOT & X1_NOT & Q1_NOT & Q0_NOT;

g21 : ANDm
     port map( Y1_in7 , Y1_7 );

     OR_Y1in <= Y1_0 & Y1_1 & Y1_2 & Y1_3 & Y1_4 & Y1_5 & Y1_6 & Y1_7;
g22 : ORm
     port map( OR_Y1in , Y1 );

     -- Following combinational logic generates circuit output(s).

     -- The following combo logic generates: out_2

     out_2_in0 <= Q1_NOT & Q0 & X2_NOT & X1_NOT;

g23 : ANDm
     port map( out_2_in0 , out_2 );

     -- The following combo logic generates: out_1

     out_1_in0 <= Q1 & Q0_NOT & X2 & X1;

g24 : ANDm
     port map( out_1_in0 , out_1 );


     -- Flip-Flops

FF1 : D_ff
          port map ( Y1, Q1, CLOCK, CLEAR, SET );

FF0 : D_ff
          port map ( Y0, Q0, CLOCK, CLEAR, SET );


     -- Initialize/Reset control

     Qinit <= "00" after 1ns,
          "ZZ" after 5ns;

     Q0 <= Qinit(0);
     Q1 <= Qinit(0);


end STRUCTURAL;
```

D.4.4 An Exercise

As an example of the verification process, perform the following exercise. This exercise performs a verification of the sequential circuit "example," which is described via the behavioral VHDL model against another sequential circuit "hand_made," which is described via the structural VHDL model. The structural VHDL description of handmade follows the exercise. All instructions are performed at the Unix prompt.

(1) type: `b2s example.vhd`

This instructs b2s to translate the VHDL behavioral circuit into a structural equivalent.

(2) type: `pre_verif -enum -vhdl example.vhd.struc`

This instructs pre_verif to process the structural description in example.vhd.struc and place its output into verif.input.

(3) type: `mv verif.input example.verif`

(4) type: `pre_verif -enum -vhdl hand_made.vhd`

This instructs pre_verif to process the structural description in hand_made.vhd and place its output into verif.input.

(5) type: `verif example.verif verif.input`

When verif execution is complete, the following should be displayed on the screen:

```
# Machine 1 inputs  2 outputs 2 latches 2
# Machine 2 inputs  2 outputs 2 latches 2
#Time to read in covers : 1.300000e-01 secs
#MACHINES ARE THE SAME
Number of states = 4
Number of edges  = 15
Number of entries  = 7
Number of save_difs  = 0
#Time for verification  : 7.000000e-02 secs
#Total user time        : 2.000000e-01 secs
```

The handmade sequential circuit is described as:

```
use work.example_pkg.all;

use work.BASICDEFS.all;


entity example is

      port(
            X1          : in  logic_mv := 'U';
            X2          : in  logic_mv := 'U';
            out_1       : out logic_mv := 'U';
            out_2       : out logic_mv := 'U';
            INITIALIZE  : in  logic_mv;
            CLOCK       : in  logic_mv

            );

end example;


architecture Hand_made of example is

 component inverter
      generic( propagation_delay : time := 0 ns );
      port (
            in1   : in logic_mv     := 'U';
            out1  : out logic_mv    := 'U'
            );

  end component;

      for all : inverter use
                  entity WORK.inverter(BEHAVIORAL);


 component ANDm
      generic( propagation_delay : time := 0 ns );
      port (
            In1   : in logic_vector_mv;
            out1  : out logic_mv    := 'U'
            );

  end component;

      for all : ANDm use
                  entity WORK.ANDm(BEHAVIORAL);


 component ORm
      generic( propagation_delay : time := 0 ns );
      port (
            In1   : in logic_vector_mv;
            out1  : out logic_mv    := 'U'
            );

  end component;
```

```
      for all : ORm use
                  entity WORK.ORm(BEHAVIORAL);


  component D_ff
      generic( propagation_delay : time := 0 ns );
      port (
            D_in  : in logic_mv     := 'U';
            Q_out       : out logic_mv    := 'U';
            CLK   : in logic_mv     := 'U';
            Clear : in logic_mv     := 'U';
            SET   : in logic_mv     := 'U'
            );

   end component;

      for all : D_ff use
                  entity WORK.D_ff(BEHAVIORAL);

      signal X1_NOT, X2_NOT, Q0_NOT, Q1_NOT : logic_mv := 'U';
      signal g4in : logic_vector_mv ( 1 downto 0 );
      signal g4out : logic_mv := 'U';
      signal g5in : logic_vector_mv ( 1 downto 0 );
      signal g5out : logic_mv := 'U';
      signal g6in : logic_vector_mv ( 2 downto 0 );
      signal g6out : logic_mv := 'U';
      signal g7in : logic_vector_mv ( 2 downto 0 );
      signal g7out : logic_mv := 'U';
      signal g8in : logic_vector_mv ( 2 downto 0 );
      signal g8out : logic_mv := 'U';
      signal g9in : logic_vector_mv ( 4 downto 0 );

      signal g10in : logic_vector_mv ( 1 downto 0 );
      signal g10out : logic_mv := 'U';
      signal g11in : logic_vector_mv ( 1 downto 0 );
      signal g11out : logic_mv := 'U';
      signal g12in : logic_vector_mv ( 2 downto 0 );
      signal g12out : logic_mv := 'U';
      signal g13in : logic_vector_mv ( 2 downto 0 );
      signal g13out : logic_mv := 'U';
      signal g14in : logic_vector_mv ( 3 downto 0 );

      signal g15in : logic_vector_mv ( 3 downto 0 );
      signal g16in : logic_vector_mv ( 3 downto 0 );

begin

   g0 : inverter
            port map( X1, X1_NOT );

   g1 : inverter
            port map( X2, X2_NOT );

   g2 : inverter
            port map( Q1, Q1_NOT );

   g3 : inverter
```

```
            port map( Q0, Q0_NOT );

      -- The following generates Y0:

      g4in <= X1 & Q0;

g4 : ANDm
            port map ( g4in , g4out );

      g5in <= X1 & Q1;

g5 : ANDm
            port map ( g5in , g5out );

      g6in <= X2_NOT & Q0 & Q1;

g6 : ANDm
            port map ( g6in , g6out );

      g7in <= X1 & X2 & Q1_NOT;

g7 : ANDm
            port map ( g7in , g7out );

      g8in <= X2 & Q0 & Q1_NOT;

g8 : ANDm
            port map ( g8in , g8out );

      g9in <= g4out & g5out & g6out & g7out & g8out;

g9 : ORm
            port map ( g9in, Y0 );

      -- The following generates Y1:

      g10in <= X2_NOT & X1_NOT;

g10 : ANDm
            port map ( g10in , g10out );

      g11in <= X1_NOT & Q0;

g11 : ANDm
            port map ( g11in , g11out );

      g12in <= X2_NOT & Q0_NOT & Q1;

g12 : ANDm
            port map ( g12in , g12out );

      g13in <= X2 & Q0 & Q1;

g13 : ANDm
            port map ( g13in , g13out );


      g14in <= g10out & g11out & g12out & g13out;
```

```
g14 : ORm
            port map ( g14in, Y1 );

      -- The following generates OUT_1:

      g15in <= X1 & X2 & Q1 & Q0_NOT;

g15 : ANDm
            port map ( g15in , out_1 );

      -- The following generates OUT_2:

      g16in <= X1_NOT & X2_NOT & Q1_NOT & Q0;

g16 : ANDm
            port map ( g16in , out_2 );

      -- The flip=flops:

 FF1 : D_ff
            port map ( Y1, Q1, CLOCK, CLEAR, SET );

 FF0 : D_ff
            port map ( Y0, Q0, CLOCK, CLEAR, SET );


      -- Initialize/Reset control

      Qinit <= "00" after 1ns,
            "ZZ" after 5ns;

      Q0 <= Qinit(0);
      Q1 <= Qinit(0);


end Hand_made;
```

Appendix E. Behavioral Design Example

This appendix contains examples of VHDL code segments representing various portions of a tm-bus module implemented using the behavioral model proposed in Chapter 3. These code segments represent implementations of a skeletal tm-bus module, the tm-bus transition process, and two representative tm-bus module states. They may be acquired by contacting:

Major K. Kanzaki
AFIT/ENG
Department of Electrical and Computer Science
School of Engineering
Air Force Institute of Technology
Wright-Patterson AFB, Ohio, 45433

Appendix F. Structural Design Examples

This appendix contains examples of sequential circuits implemented using the structural model proposed in Chapter 3. See Chapter 5 for details concerning the functionality of these sequential circuits.

```
use WORK.basicdefs.all;
use work.SD_package.all;
entity Sequence_Detector is
      -- generic ( );

      port (
            Xin         : in logic_mv     := 'U';
            CLOCK       : in logic_mv     := 'U';
            Zout        : out logic_mv    := 'U';
            INITIALIZE  : in logic_mv     := 'U'
           );
end Sequence_Detector;


architecture STRUCTURAL1 of Sequence_Detector is

 component inverter
      generic( propagation_delay : time := 0 ns );
      port (
            in1   : in logic_mv     := 'U';
            out1  : out logic_mv    := 'U'
            );
  end component;

      for all : inverter use
                  entity WORK.inverter(BEHAVIORAL);

  component ANDm
        generic( propagation_delay : time := 0 ns );
        port(
             In1      : in logic_vector_mv;
             out1     : out logic_mv := 'U'
               );
  end component;

      for all : ANDm use
                  entity WORK.ANDm(BEHAVIORAL);

  component ORm
        generic( propagation_delay : time := 0 ns );
        port(
             In1      : in logic_vector_mv;
             out1     : out logic_mv := 'U'
               );
  end component;

      for all : ORm use
                  entity WORK.ORm(BEHAVIORAL);

  component D_ff
      generic ( propagation_delay : time  := 0 ns );
      port (
            D_in  : in logic_mv     := 'U';
            Q_out : out logic_mv    := 'U';
            CLK   : in logic_mv     := 'U';
            Clear : in logic_mv     := 'U';
            SET   : in logic_mv     := 'U'
            );
```

```
  end component;

      for all : d_ff use
                  entity WORK.D_ff(BEHAVIORAL);

  --- Internal Signal Declarations

      signal  Y1,
              Y2,
              Q1,
              Q2 : logic_mv := 'U';

      signal  Xnot,
              Q1not,
              Q2not,
              AND1_output : logic_mv := 'U';

      signal  AND1_input,
              AND2_input,
              OR1_input,
              Qinit  : logic_vector_mv (1 downto 0) := "UU";

      signal  AND3_input : logic_vector_mv (2 downto 0)
                                 := "UUU";

begin

  g1 : inverter
              port map (Q2, Q2not);

  g2 : inverter
              port map (Xin, Xnot);

  g3 : inverter
              port map (Q1, Q1not);

  ---  Logic to derive Y1:

      AND1_input <= Q1 & Q2not;

      g4 : ANDm
              port map ( AND1_input, AND1_output );

      OR1_input <= Xin & AND1_output;

      g5 : ORm
              port map ( OR1_input, Y1 );

  --- LOGIC to derive Y2

      AND2_input <= Xnot & Q1 ;

      g6 : ANDm
              port map ( AND2_input, Y2 );

  --- LOGIC to derive Zout

      AND3_input <= Xin & Q1not & Q2 ;
```

```
      g7 : ANDm
            port map ( AND3_input, Zout );

  --- Registers

  FF1 : D_ff
            port map ( Y1, Q1, CLOCK, CLEAR, SET );

  FF2 : D_ff
            port map ( Y2, Q2, CLOCK, CLEAR, SET  );


Qinit  <=   "00" after 1ns,
      "ZZ" after 20ns;
      q1 <= Qinit(0);
      q2 <= Qinit(1);


end STRUCTURAL1;
```

```
name Sequence_Detector1
i xin
o Zout


g1 not  q2 ; q2not

g2 not  xin ; xnot

g3 not  q1 ; q1not


#  ---  Logic to derive Y1:

g4 and  q1  q2not ; AND1out

g5 or   xin  AND1out ;  Y1


# --- LOGIC to derive Y2

g6 and  xnot q1 ; Y2


#  --- LOGIC to derive zout

g7 and  xin q1not q2 ; Zout

# --- Registers

ps q1
ns Y1

ps q2
ns Y2

I
00
```

```
use WORK.basicdefs.all;

entity Sequence_Detector is
      -- generic ( );

      port (
            Xin          : in logic_mv     := 'U';
            CLOCK : in logic_mv     := 'U';
            Zout         : out logic_mv    := 'U';
            INITIALIZE   : in logic_mv     := 'U'
           );
end Sequence_Detector;


architecture STRUCTURAL2 of Sequence_Detector  is

 component inverter
      generic( propagation_delay : time := 0 ns );
      port (
            in1    : in logic_mv     := 'U';
            out1   : out logic_mv    := 'U'
            );
  end component;

      for all : inverter use
                  entity WORK.inverter(BEHAVIORAL);

  component NANDm
        generic( propagation_delay : time := 0 ns );
        port(
             In1   : in logic_vector_mv;
             out1     : out logic_mv := 'U'
                );
  end component;

      for all : NANDm use
                  entity WORK.NANDm(BEHAVIORAL);

  component ORm
        generic( propagation_delay : time := 0 ns );
        port(
             In1   : in logic_vector_mv;
             out1     : out logic_mv := 'U'
                );
  end component;

      for all : ORm use
                  entity WORK.ORm(BEHAVIORAL);



  component D_ff
      generic ( propagation_delay    : time := 0 ns );
      port (
            D_in   : in logic_mv      := 'U';
            Q_out  : out logic_mv     := 'U';
            CLK    : in logic_mv      := 'U';
            Clear  : in logic_mv      := 'U';
```

```
            SET    : in logic_mv      := 'U'
            );
    end component;

        for all : d_ff use
                    entity WORK.D_ff(BEHAVIORAL);

    --- Internal Signal Declarations

        signal  Y1,
                Y2,
                Q1,
                Q2 : logic_mv := 'U';

        signal  Xnot,
                Q1not,
                Q2not,
                NAND1_output,
                NAND2_output,
                NAND3_output,
                NAND4_output : logic_mv := 'U';

        signal  NAND1_input,
                NAND2_input,
                NAND3_input,
                OR1_input,
                Qinit  : logic_vector_mv (1 downto 0) := "UU";

        signal  NAND4_input : logic_vector_mv (2 downto 0) := "UUU";

begin

    g1 : inverter
            port map (Xin, Xnot);

    g2 : inverter
            port map (Q2, Q2not);

    g3 : inverter
            port map (Q1, Q1not);

    ---  Logic to derive Y1:

        NAND1_input <= Q1 & Q2not;

        g4 : NANDm
            port map ( NAND1_input, NAND1_output );

        NAND2_input <= Xnot & NAND1_output;

        g5 : NANDm
            port map ( NAND2_input, Y1 );

    --- LOGIC to derive Y2

        NAND3_input <= Xnot & Q1;

        g6 : NANDm
```

```
            port map ( NAND3_input, NAND3_output );

        g7 : inverter
            port map ( NAND3_output, Y2 );

    --- LOGIC to derive Zout

        NAND4_input <= Xin & Q1not & Q2;

        g8 : NANDm
            port map ( NAND4_input, NAND4_output );

        g9  : inverter
            port map ( NAND4_output , Zout );

    --- Registers

    FF1 : D_ff
            port map ( Y1, Q1, CLOCK, CLEAR, SET );

    FF2 : D_ff
            port map ( Y2, Q2, CLOCK, CLEAR, SET  );


-- The following specifies the initial state vector of the register:

Qinit  <=   "00" after 1ns,
      "ZZ" after 20ns;
      q1 <= Qinit(0);
      q2 <= Qinit(1);

end STRUCTURAL2;
```

```
name Sequence_Detector
i Xin
o Zout

g1 not Xin ; Xnot

g2 not Q2 ; Q2not

g3 not Q1 ; Q1not

# ---  Logic to derive Y1:

g4 nand  Q1  Q2not ; NAND1_output

g5 nand  Xnot NAND1_output ;  Y1

#  --- LOGIC to derive Y2

g6 nand  Xnot  Q1 ;  NAND3_output

g7 not   NAND3_output ;  Y2

#  --- LOGIC to derive Zout

g8 nand      Xin  Q1not  Q2 ; NAND4_output

g9 not       NAND4_output ; Zout

#  --- Registers

ps Q1
ns Y1

ps Q2
ns Y2

I
00
```

```
use WORK.basicdefs.all;

entity Sequence_Detector is

      port (
            Xin          : in logic_mv      := 'U';
            CLOCK        : in logic_mv      := 'U';
            Zout         : out logic_mv     := 'U';
            INITIALIZE   : in logic_mv      := 'U'
           );
end Sequence_Detector;


architecture STRUCTURAL3 of Sequence_Detector is

 component inverter
      generic( propagation_delay : time := 0 ns );
      port (
            in1    : in logic_mv      := 'U';
            out1   : out logic_mv     := 'U'
            );
  end component;

      for all : inverter use
                  entity WORK.inverter(BEHAVIORAL);

  component ANDm
        generic( propagation_delay : time := 0 ns );
        port(
             In1          : in logic_vector_mv;
             out1     : out logic_mv := 'U'
                );
  end component;

      for all : ANDm use
                  entity WORK.ANDm(BEHAVIORAL);

  component ORm
        generic( propagation_delay : time := 0 ns );
        port(
             In1          : in logic_vector_mv;
             out1     : out logic_mv := 'U'
                );
  end component;

      for all : ORm use
                  entity WORK.ORm(BEHAVIORAL);

  component D_ff
      generic ( propagation_delay : time  := 0 ns );
      port (
            D_in   : in logic_mv      := 'U';
            Q_out  : out logic_mv     := 'U';
            CLK    : in logic_mv      := 'U';
            Clear  : in logic_mv      := 'U';
            SET    : in logic_mv      := 'U'
            );
  end component;
```

```
      for all : d_ff use
                  entity WORK.D_ff(BEHAVIORAL);

  --- Internal Signal Declarations

      signal  Y1,
              Y2,
              Q1,
              Q2 : logic_mv := 'X';

      signal  Xnot,
              Q1not,
              Q2not,
              AND1_output,
              OR1_output  : logic_mv := 'X';

      signal  AND1_input,
              AND2_input,
              OR1_input,
              Qinit  : logic_vector_mv (1 downto 0) := "XX";

      signal  AND3_input : logic_vector_mv (2 downto 0)
                               := "XXX";

begin

--  g1 : inverter
--          port map (Q2, Q2not);

  g2 : inverter
          port map (Xin, Xnot);

  g3 : inverter
          port map (Q1, Q1not);


  --- LOGIC to derive Y1
  ---       where Y2 = xnot and ( q1 or q2 )

      OR1_input <= Q1 & Q2;

      g4  : ORm
            port map ( OR1_input, OR1_output );

      AND2_input <= Xnot & OR1_output;

      g5 : ANDm
            port map ( AND2_input, Y1 );

  ---  Logic to derive Y1
  ---       where Y2 = xnot and q1not

      AND1_input <= Xnot & Q1not;

      g6 : ANDm
            port map ( AND1_input, Y2 );
```

```
  --- LOGIC to derive Zout

      AND3_input <= Xin & Q1 & Q2;

      g7 : ANDm
           port map ( AND3_input, Zout );

  --- Registers


  FF1 : D_ff
           port map ( Y1, Q1, CLOCK, CLEAR, SET );

  FF2 : D_ff
           port map ( Y2, Q2, CLOCK, CLEAR, SET  );

when INITIALIZE select
     Qinit <= "01" when '1',
     Qinit <= "ZZ" when others;

end STRUCTURAL3;
```

```
name Sequence_Detector
i Xin
o Zout

g1 not Q2 ; Q2not

g2 not Xin ; Xnot

g3 not Q1 ; Q1not


#  --- LOGIC to derive Y1
#  ---             where Y2 = xnot and ( q1 or q2 )

g4 or   Q1  Q2 ; OR1_output

g5 and  Xnot  OR1_output ; Y1

#  ---  Logic to derive Y1
#  ---             where Y2 = xnot and q1not

g6 and Xnot  Q1not ; Y2


#  --- LOGIC to derive Zout

g7 and  Xin  Q1  Q2 ; Zout

ps Q1
ns Y1

ps Q2
ns Y2

I
01
```

```
entity TEST_BENCH is
end TEST_BENCH;

use work.BASICDEFS.all;
--use WORK.SD_Package.all;
use STD.SIMULATOR_STANDARD.all;
use STD.TEXTIO.all;

architecture SD_test of TEST_BENCH is

  component Sequence_Detector

      port (
            Xin   : in logic_mv     := 'U';
            CLOCK : in logic_mv     := 'U';
            Zout  : out logic_mv    := 'U';
            Clear : in logic_mv     := 'U';
            Set   : in logic_mv     := 'U';
           );
  end component;

      for UUT1 : Sequence_Detector use
                  entity work.Sequence_Detector(Structural1);

      for UUT2 : Sequence_Detector use
                  entity WORK.Sequence_Detector(structural2);

      for UUT3 : Sequence_Detector use
                  entity WORK.Sequence_Detector(structural3);



  signal instruction      :   logic_vector_mv (2 downto 0)
                          := "UUU";
  alias input_string      :   logic_mv is instruction(0);
  alias RESET             :   logic_mv is instruction(1);
  alias CLEAR             :   logic_mv is instruction(2);

  signal   CLOCK         :     logic_mv;
  signal   OUT_1,
           OUT_2,
           OUT_3         :     logic_mv;

begin


process
    file INSTRUCTIONS  : TEXT is in "Input_String";
    variable L     : Line;
    variable machine_code : bit_vector (2 downto 0);
    begin
      readline(INSTRUCTIONS, L);
            if ENDFILE(INSTRUCTIONS) then terminate;    : if;
      read(L, machine_code);
            case machine_code is
                  when "000" => instruction <= "000";
                  when "001" => instruction <= "001";
                  when "010" => instruction <= "010";
```

```
                when "011" => instruction <= "011";
                when "100" => instruction <= "100";
                when "101" => instruction <= "101";
                when "110" => instruction <= "110";
                when "111" => instruction <= "111";
            end case;
      wait for 8ns;
    end process;



 process
    begin
      set_maximums(10000,100);
      tracing_on;
      wait for 500ns;
      terminate;
    end process;


  make_Clock : process
    begin
      wait for 2ns;
       CLOCK <= '1';
       wait for 4ns;
       CLOCK <= '0';
      wait for 2ns;
    end process make_Clock;

   UUT1 : Sequence_Detector
          port map (input_string, CLOCK, OUT_1, RESET, CLEAR);

   UUT2 : Sequence_Detector
          port map (input_string, CLOCK, OUT_2, RESET, CLEAR);

   UUT3 : Sequence_Detector
          port map (input_string, CLOCK, OUT_3, RESET, CLEAR);

end SD_test;
```

Vhdl Simulation Report

Report Name: SD_test"
Kernel Libra.y Name: <<RMILLER.STRUCTURAL>>SD_TEST
Kernel Creation Date: AUG-13-1990
Kernel Creation Time: 00:11:03
Run Identifer: 1
Run Date: AUG-13-1990
Run Time: 00:11:03

Report Control Language File: test.rcl
Report Output File : sd_test.rpt

Max Time: 9223372036854775807
Max Delta: 2147483646

Report Control Language :

```
Simulation_report TEST is
begin
report_name is "SD_test";
page_width is 120;
page_length is 60;
signal_format is horizontal;

sample_signals by_transaction in ns;
--sample_signals by_event in ns;
select_signal : Clock;
--select_signal : reset;
select_signal : instruction;
select_signal /uut1: Q2;
select_signal /uut1: Q1;
select_signal : out_1;
select_signal /uut2: Q2;
select_signal /uut2: Q1;
select_signal : out_2;
select_signal /uut3: Q2;
select_signal /uut3: Q1;
select_signal : out_3;

end TEST;
```

```
Report Format Information :

  Time is in NS relative to the start of simulation
  Time period for report is from 0 NS to End of Simulation
  Signal values are reported by transaction  ( ' ' indicates no transaction )
```

SD_test"

TIME	SIGNAL NAMES										
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	Q2	Q1	OUT_1	Q2	Q1	OUT_2	Q2	Q1	OUT_3
0	'U'	"UUU"	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
+1		"100"			'X'			'X'			'X'
+2								'X'			'X'
+3					'0'						'0'
+4								'0'			
2											
+1	'1'										
+2			'0'	'0'		'0'	'0'		'0'	'1'	
+4					'0'						'0'
+5					'0'						
6											
+1	'0'										
8											
+1		"000"									
10											
+1	'1'										
+2			'0'	'0'		'0'	'0'		'0'	'1'	
14											
+1	'0'										
16											
+1		"000"									
18											
+1	'1'										
+2			'0'	'0'		'0'	'0'		'0'	'1'	
22											
+1	'0'										
24											
+1		"001"									
+3					'0'						'0'
26											
+1	'1'										
+2			'0'	'1'		'0'	'1'		'0'	'0'	
+4											'0'
+5					'0'						
30											
+1	'0'										

TIME	SIGNAL NAMES										
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	Q2	Q1	OUT_1	Q2	Q1	OUT_2	Q2	Q1	OUT_3
32											
+1		"000"									
+3					'0'						'0'
34											
+1	'1'										
+2			'1'	'1'		'1'	'1'		'1'	'0'	
+4					'0'						'0'
38											
+1	'0'										
40											
+1		"000"									
42											
+1	'1'										
+2			'1'	'0'		'1'	'0'		'1'	'1'	
+4											'0'
+5					'0'						
46											
+1	'0'										
48											
+1		"001"									
+3					'1'						'1'
+4								'1'			
50											
+1	'1'										
+2			'0'	'1'		'0'	'1'		'0'	'0'	
+4					'0'						'0'
+5					'0'			'0'			
54											
+1	'0'										
56											
+1		"000"									
+3					'0'						'0'
58											
+1	'1'										
+2			'1'	'1'		'1'	'1'		'1'	'0'	
+4					'0'						'0'
62											
+1	'0'										

TIME	SIGNAL NAMES										
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	Q2	Q1	OUT_1	Q2	Q1	OUT_2	Q2	Q1	OUT_3
64											
+1		"001"									
+3					'0'						'0'
66											
+1	'1'										
+2			'0'	'1'		'0'	'1'		'0'	'0'	
+4					'0'						'0'
70											
+1	'0'										
72											
+1		"001"									
74											
+1	'1'										
+2			'0'	'1'		'0'	'1'		'0'	'0'	
78											
+1	'0'										
80											
+1		"001"									
82											
+1	'1'										
+2			'0'	'1'		'0'	'1'		'0'	'0'	
86											
+1	'0'										
88											
+1		"000"									
+3					'0'						'0'
90											
+1	'1'										
+2			'1'	'1'		'1'	'1'		'1'	'0'	
+4					'0'						'0'
94											
+1	'0'										
96											
+1		"000"									

TIME	SIGNAL NAMES										
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	Q2	Q1	OUT_1	Q2	Q1	OUT_2	Q2	Q1	OUT_3
98											
+1	'1'										
+2			'1'	'0'		'1'	'0'		'1'	'1'	
+4											'0'
+5					'0'						
102											
+1	'0'										
104											
+1		"011"									
+3					'1'						'1'
+4								'1'			
106											
+1	'1'										
+2			'1'	'1'		'1'	'1'		'1'	'0'	
+4											'0'
+5					'0'						
+6								'0'			
110											
+1	'0'										

```
use work.BASICDEFS.all;
use work.CPU_package.all;

entity CPU_CONTROLLER1 is
      -- generic ();
      port( INSTR2       : in logic_mv;
            INSTR1       : in logic_mv;
            INSTR0       : in logic_mv;
            CLOCK        : in logic_mv;
            RESET        : in logic_mv     ;
            ZERO_FLAG    : in logic_mv;
            Control_bus : out logic_vector_mv_bus (12 downto 0)
                         := "XXXXXXXXXXXXX";
            INITIALIZE   : in logic_mv
            );
end;


architecture STRUCTURAL of CPU_CONTROLLER1 is

 component inverter
      generic( propagation_delay : time := 0 ns );
      port (
            in1    : in logic_mv      := 'U';
            out1   : out logic_mv     := 'U'
            );
  end component;

      for all : inverter use
                  entity WORK.inverter(BEHAVIORAL);

  component ANDm
        generic( propagation_delay : time := 0 ns );
        port(In1      : in logic_vector_mv;
             out1     : out logic_mv := 'U'
                );
  end component;

      for all : ANDm use
                  entity WORK.ANDm(BEHAVIORAL);

  component ORm
        generic( propagation_delay : time := 0 ns );
        port(In1      : in logic_vector_mv;
             out1     : out logic_mv := 'U'
                );
  end component;

      for all : ORm use
                  entity WORK.ORm(BEHAVIORAL);

  component D_ff
      generic ( propagation_delay : time  := 0 ns );
      port (
            D_in   : in logic_mv      := 'U';
            Q_out  : out logic_mv     := 'U';
            CLK    : in logic_mv      := 'U';
            Clear  : in logic_mv      := 'U';
```

```
            SET    : in logic_mv      := 'U'
            );
    end component;

        for all : d_ff use
                    entity WORK.D_ff(BEHAVIORAL);

    --- Internal Signal Declarations

        signal INSTR2not, INSTR1not,
               INSTR0not, ZERO_FLAGnot : logic_mv := 'U';


        signal  Y0, Y1, Y2, Y3, Y4, Y5, Y6,
                Y7, Y8, Y9, Y10, Y11, Y12, YR0, YI0 : logic_mv := 'U';

        signal  Q0, Q1, Q2, Q3, Q4, Q5, Q6,
                Q7, Q8, Q9, Q10, Q11, Q12,
                QR0, QI0 : Wired_OR logic_mv := 'U';

        signal  Q0not, Q1not, Q2not, Q3not, Q4not, Q5not, Q6not,
                Q7not, Q8not, Q9not, Q10not, Q11not, Q12not,
                QR0not, QI0not                    : logic_mv := 'U';


        signal ANDyr0 : logic_mv_vector(1 downto 0) := "UU";

        signal ANDload, ANDstore, ANDadd, ANDand,
               ANDjump, ANDjumpz, ANDcomp,
               ANDrshift, : logic_vector_mv(2 downto 0) := "UUU";

        signal ANDy2, ANDy3, ORy3b, ANDy4, ANDy5,
               ANDyi0, ORyi0, ANDy7, ANDy8a, ORy8,
               ANDy9, ANDy10b, ANDy12, ANDyi0,
               ORyi0, ANDyr0 : logic_vector_mv(1 downto 0) := "UU";

        signal ANDy0, ANDy1, ORy3a, ANDy6, ANDy8b, ANDy10a,
               ANDy11, ORyr0     : logic_vector_mv(2 downto 0) := "UUU";

        signal ORy7 : logic_vector_mv(3 downto 0) := "UUUU";

        signal ORY10  : logic_vector_mv(7 downto 0) := "UUUUUUUU";

        signal CLEAR, SET : logic_mv := '0';


begin

    g1: inverter
         port map ( Q0, Q0not );

    g2: inverter
         port map ( Q1, Q1not );

    g3: inverter
         port map ( Q2, Q2not );

    g4: inverter
```

```
      port map ( Q3, Q3not );

 g5: inverter
      port map ( Q4, Q4not );

 g6: inverter
      port map ( Q5, Q5not );

 g7: inverter
      port map ( Q6, Q6not );

 g8: inverter
      port map ( Q7, Q7not );

 g9: inverter
      port map ( Q8, Q8not );

g10: inverter
      port map ( Q9, Q9not );

g11: inverter
      port map ( Q10, Q10not );

g12: inverter
      port map ( Q11, Q11not );

g13: inverter
      port map ( Q12, Q12not );

g14: inverter
      port map ( QR0, QR0not );

g15: inverter
      port map ( QI0, QI0not );

g16: inverter
      port map (INSTR2, INSTR2not );

g17: inverter
      port map (INSTR1, INSTR1not );

g18: inverter
      port map (INSTR0, INSTR0not );

gZFnot: inverter
      port map ( ZERO_FLAG, ZERO_FLAGnot);

-------------------------------------------------------------------------
--    Following code decodes instructions                              --
-------------------------------------------------------------------------


-- logic to decode LOAD instruction
-- LOAD = INSTR2not and INSTR1not and INSTR0not;

      ANDload <= INSTR2not & INSTR1not & INSTR0not;

gLOAD: ANDm
```

```
      port map ( ANDload , LOAD );

-- logic to decode STORE instruction
-- STORE = INSTR2not and INSTR1not and INSTR0;

      ANDstore <= INSTR2not & INSTR1not & INSTR0;

gSTORE: ANDm
      port map ( ANDstore, STORE );

-- logic to decode ADD instruction
-- ADD = INSTR2not and INSTR1 and INSTR0not;

      ANDadd <= INSTR2not & INSTR1 & INSTR0not;

gADD: ANDm
      port map( ANDadd, ADD);

-- logic to decode AND instruction
-- AND = INSTR2not and INSTR1 and INSTR0;

      ANDand <= INSTR2not & INSTR1 & INSTR0;

gAND: ANDm
      port map( ANDand, AND);

-- logic to decode JUMP instruction
-- JUMP = INSTR2 and INSTR1not and INSTR0not;

      ANDjump <= INSTR2 & INSTR1not & INSTR0not;

gJUMP : ANDm
      port map( ANDjump, JUMP );

-- logic to decode JUMPZ instruction
-- JUMPZ = INSTR2 and INSTR1not and INSTR0;

      ANDjumpz <= INSTR2 & INSTR1not & INSTR0;

gJUMPZ : ANDm
      port map( ANDjump, JUMPZ );

--  ogic to decode COMP instruction
-- COMP = INSTR2 and INSTR1not and INSTR0not;

      ANDcomp <= INSTR2 & INSTR1 & INSTR0not;

gCOMP: ANDm
      port map( ANDcomp, COMP );

-- logic to decode RSHIFT instruction
-- RSHIFT = INSTR2 and INSTR1not and INSTR0not;

      ANDrshift <= INSTR2 & INSTR1 & INSTR0;

gRSHIFT: ANDm
      port map( ANDrshift, RSHIFT );
```

```
------------------------------------------------------------------------
--    Following code derives Next states                               --
------------------------------------------------------------------------

-- Logic to derive Y0
-- Y0 = q3 and qR0 and Qnot and ADD

      ANDy0 <= Q3 & QR0 & ADD;

g19: ANDm
      port map ( ANDy0, Y0 );

-- Logic to derive Y1
-- Y1 = Q3 and QR0 and AND

      ANDy1 <= Q3 & QR0 & AND;

g20: ANDm
      port map ( ANDy1, Y1 );

-- Logic to derive Y2
-- Y2 = Q11 and COMP

      ANDy2 <= Q11 & COMP;

g21: ANDm
      port map ( ANDy2, Y2 );

-- Logic to derive Y3
-- Y3 = [(LOAD or ADD or AND) and q7] or
--       [q10]

      ORy3a <= LOAD & ADD & AND;

g21: ORm
      port map ( ORy3a, ORy3aout);


      ANDy3 <= Q7 & ORy3aout;

g22: ANDm
      port map( ANDy3, ANDy3out);

      ORy3b <= Q10 & ANDy3out;

g23: ORm
      port map (ORy3b, Y3);

-- Logic to derive Y4
-- Y4 = q5 and STORE

      ANDy4 <= Q5 & STORE;

g24: ANDm
      port map (ANDy4, Y4);

-- logic to derive Y5
-- Y5 = q7 and STORE
```

```
      ANDy5 <= Q7 & STORE;

g25: ANDm
      port map( ANDy5, Y5 );

-- logic to derive YI0
-- YI0 = q3 or (q4 and STORE)

      ANDyi0 <= Q4 & STORE;

g26: ANDm
      port map ( ANDyi0, ANDyi0out);

      ORyi0 <= ANDyi0out & q3;

g27: ORm
      port map( ORyi0, YI0);

-- Logic to derive Y6
-- Y6 = q3 and qR0 and LOAD

      ANDy6 <= Q3 & QR0 & LOAD;

g28: ANDm
      port map( ANDy6, Y6 );

-- Logic to derive Y7
-- Y7 = (load or store or add or and) and q11

      ORy7 <= LOAD & STORE & ADD & AND;

g29: ORm
      port map ( ORy7, ORy7out);

      ANDy7 <= ORy7out & Q11;

g30: ANDm
      port map ( ANDy7, Y7);

-- Logic to derive Y8
-- Y8 = (q11 and JUMP) or (q11 and JUMPZ and ZERO_FLAGnot)


      ANDy8a <= Q11 & JUMP;
      ANDy8b <= Q11 & JUMPZ & ZERO_FLAGnot;
      ORy8 <= ANDy8aout & ANDy8bout;

g31: ANDm
      port map ( ANDy8a, ANDy8aout );

g32: ANDm
      port map ( ANDy8b, ANDy8bout );

g33: ORm
      port map ( ORy8, Y8 );

-- Logic to derive Y9
```

```
-- Y9 = q3 & Qnot

      ANDy9 <= Q3 & QI0not;

g34: ANDm
      port map ( ANDy9, Y9 );

-- Logic to derive Y10
-- Y10 = (JUMPZ and Zero_Flagnot and Q11) or
--       q6 or q0 or q1 or (q4 and qI0) or q12 or q3 or q8

      ANDy10a <= JUMPZ & ZERO_FLAGnot & Q11;
      ANDy10b <= Q4 & QI0;
      ORY10 <= ANDy10aout & ANDy10bout & Q6 & Q0 & Q1 & Q12 & Q3 & Q8;

g35: ANDm
      port map ( ANDy10a, ANDy10aout);

g36: ANDm
      port map ( ANDy10b, ANDy10bout);

g37: ORm
      port map ( ORY10, Y10);

-- Logic to derive Y11
-- Y11 = q3 and q9 and qI0

      ANDy11 <= Q3 & Q9 & QI0;

g38: ANDm
      port map ( ANDy11, Y11);

-- Logic to derive Y12
-- Y12 = q11 and rshift

      ANDy12 <= Q11 & RSHIFT;

g39: ANDm
      port map ( ANDy12 , Y12);

-- Logic to derive YI0
-- YI0 = q3 or (q4 and STORE)


      ANDyi0 <= Q4 & STORE;
      ORyi0 <= Q3 & ANDyi0out;

g40: ANDm
      port map ( ANDyi0, ANDyi0out);

g41: ORm
      port map ( ORyi0, YI0);

-- Logic to derive YR0
-- YR0 = q3 and (load or add or and)

      ORyr0 <= LOAD & ADD & AND;
```

```
g42: ORm
      port map ( ORyr0, ORyr0out);

      ANDyr0 <= ORyr0out & Q3;

g43: ANDm
      port map ( ANDyr0, YR0);

--------------------------------------------------------------------------
--    Registers:
--------------------------------------------------------------------------

  FF0 : D_ff
          port map ( Y0, Q0, CLOCK, CLEAR, SET );

  FF1 : D_ff
          port map ( Y1, Q1, CLOCK, CLEAR, SET );

  FF2 : D_ff
          port map ( Y2, Q2, CLOCK, CLEAR, SET );

  FF3 : D_ff
          port map ( Y3, Q3, CLOCK, CLEAR, SET );

  FF4 : D_ff
          port map ( Y4, Q4, CLOCK, CLEAR, SET );

  FF5 : D_ff
          port map ( Y5, Q5, CLOCK, CLEAR, SET );

  FF6 : D_ff
          port map ( Y6, Q6, CLOCK, CLEAR, SET );

  FF7 : D_ff
          port map ( Y7, Q7, CLOCK, CLEAR, SET );

  FF8 : D_ff
          port map ( Y8, Q8, CLOCK, CLEAR, SET );

  FF9 : D_ff
          port map ( Y9, Q9, CLOCK, CLEAR, SET );

  FF10 : D_ff
          port map ( Y10, Q10, CLOCK, CLEAR, SET );

  FF11 : D_ff
          port map ( Y11, Q11, CLOCK, CLEAR, SET );

  FF12 : D_ff
          port map ( Y12, Q12, CLOCK, CLEAR, SET );

  FFI0 : D_ff
          port map ( YI0, QI0, CLOCK, CLEAR, SET );


  FFR0 : D_ff
          port map ( YR0, QR0, CLOCK, CLEAR, SET );
```

```
-- SET UP INITIAL STATE:

when INTIALIZE select
      Qinit <= "000010000000000" when '1',
      Qinit <= "ZZZZZZZZZZZZZZZ" when others;

       Q0 <= Qinit( 0);
       Q1 <= Qinit( 1);
       Q2 <= Qinit( 2);
       Q3 <= Qinit( 3);
       Q4 <= Qinit( 4);
       Q5 <= Qinit( 5);
       Q6 <= Qinit( 6);
       Q7 <= Qinit( 7);
       Q8 <= Qinit( 8);
       Q9 <= Qinit( 9);
      Q10 <= Qinit(10);
      Q11 <= Qinit(11);
      Q12 <= Qinit(12);
      QI0 <= Qinit(13);
      QR0 <= Qinit(14);

-- Drive outputs:

Control_bus( 0) <= Q0;
Control_bus( 1) <= Q1;
Control_bus( 2) <= Q2;
Control_bus( 3) <= Q3;
Control_bus( 4) <= Q4;
Control_bus( 5) <= Q5;
Control_bus( 6) <= Q6;
Control_bus( 7) <= Q7;
Control_bus( 8) <= Q8;
Control_bus( 9) <= Q9;
Control_bus(10) <= Q10;
Control_bus(11) <= Q11;
Control_bus(12) <= Q12;


end STRUCTURAL;
```

The following VHDL code creates an incorrect controller response to the JUMPZ instruction. This combinational logic code which derives Y10 (the input signal for D flip-flop FF10) was substituted for the correct version in the controller VHDL code to produce an incorrect controller version.

```
-- Logic to derive Y10
--  ERROR ERROR ERROR ERROR ERROR ERROR ERROR ERROR ERROR ERROR ERROR
--  Y10 should be:
--  Y10 = (JUMPZ and Zero_Flag and C11) or (c4 and qWRITEnot) or
--       c6 or c0 or c1 or c12 or c2 or c8
--  But instead it's:
--  Y10 = (JUMPZ and Zero_Flagnot and C11) or (c4 and qWRITEnot) or
--       c6 or c0 or c1 or c12 or c2 or c8

      ANDy10a <= JUMPZ & ZERO_FLAGnot & C11;
      ANDy10b <= C4 & QWRITEnot;
      ORY10 <= ANDy10aout & ANDy10bout & C6 & C0 & C1 & C12 & C2 & C8;

g35: ANDm
      port map ( ANDy10a, ANDy10aout);

g36: ANDm
      port map ( ANDy10b, ANDy10bout);

g37: ORm
      port map ( ORY10, Y10);
```

```
use work.BASICDEFS.all;
--use work.CPU_package.all;
use STD.SIMULATOR_STANDARD.all;
use STD.TEXTIO.all;


entity TEST_BENCH is
end TEST_BENCH;


architecture structural_CPU_588 of TEST_BENCH is

  component  CPU_CONTROLLER1
      -- generic ();
        port (  INSTR2   : in logic_mv;
                INSTR1   : in logic_mv;
                INSTR0   : in logic_mv;
                C12      : inout wired_outputs logic_mv := 'U';
                C11      : inout wired_outputs logic_mv := 'U';
                C10      : inout wired_outputs logic_mv := 'U';
                C9       : inout wired_outputs logic_mv := 'U';
                C8       : inout wired_outputs logic_mv := 'U';
                C7       : inout wired_outputs logic_mv := 'U';
                C6       : inout wired_outputs logic_mv := 'U';
                C5       : inout wired_outputs logic_mv := 'U';
                C4       : inout wired_outputs logic_mv := 'U';
                C3       : inout wired_outputs logic_mv := 'U';
                C2       : inout wired_outputs logic_mv := 'U';
                C1       : inout wired_outputs logic_mv := 'U';
                C0       : inout wired_outputs logic_mv := 'U';
                CLOCK            : in logic_mv;
                ZERO_FLAG        : in logic_mv;
                INITIALIZE       : in logic_mv
                );
     end component;

  for UUT1 : CPU_CONTROLLER1 use
                        entity WORK.CPU_CONTROLLER1(STRUCTURAL);
  for UUT2 : CPU_CONTROLLER1 use
                        entity WORK.CPU_CONTROLLER1(STRUCTURAL_bogus);

      signal       INITIALIZE  : logic_mv  := 'U';
      signal       CLOCK : logic_mv  := 'U';
      signal Instruction       : logic_mv_vector (2 downto 0);
      signal       Control_good,
            Control_bogus      : logic_vector_mv  := "UUUUUUUUUUUUU";


begin

process
    file CPU_INSTRUCTIONS  : TEXT is in "CPU_INSTRUCTIONS";
    variable L     : Line;
      variable temp_instruction     : bit_vector (4 downto 0);
      variable temp2_instruction    : bit_vector (4 downto 0);
```

```
    begin

     readline(CPU_INSTRUCTIONS, L);
     if ENDFILE(CPU_INSTRUCTIONS) then terminate; end if;
     read(L, temp_instruction);

      case temp_instruction is
            when '0' => Zero_Flag <= '0';
            when '1' => Zero_Flag <= '1';
      end case;

      wait until Control_good = "0000000001000";

      case temp_instruction is
            when "000" => instruction <= "000";
            when "001" => instruction <= "001";
            when "010" => instruction <= "010";
            when "011" => instruction <= "011";
            when "100" => instruction <= "100";
            when "101" => instruction <= "101";
            when "110" => instruction <= "110";
            when "111" => instruction <= "111";
      end case;
    end process;

process
    begin
      set_maximums(10000,100);
      tracing_on;
      wait for 1500ns;
      terminate;
    end process;


make_Clock : process
  begin
    wait for 2ns;
    CLOCK <= '0';
    wait for 4ns;
    CLOCK <= '1';
    wait for 2ns;
  end process make_Clock;


UUT1 : CPU_CONTROLLER1
      port map( Instruction(2),
            Instruction(1),
            Instruction(0),
            Control_good(12),
            Control_good(11),
            Control_good(10),
            Control_good( 9),
            Control_good( 8),
            Control_good( 7),
            Control_good( 6),
            Control_good( 5),
            Control_good( 4),
            Control_good( 3),
```

```
            Control_good( 2),
            Control_good( 1),
            Control_good( 0),
            CLOCK,
            ZERO_FLAG,
            INITIALIZE
            );

UUT2 : CPU_CONTROLLER1
      port map( Instruction(2),
            Instruction(1),
            Instruction(0),
            Control_bogus(12),
            Control_bogus(11),
            Control_bogus(10),
            Control_bogus( 9),
            Control_bogus( 8),
            Control_bogus( 7),
            Control_bogus( 6),
            Control_bogus( 5),
            Control_bogus( 4),
            Control_bogus( 3),
            Control_bogus( 2),
            Control_bogus( 1),
            Control_bogus( 0),
            CLOCK,
            ZERO_FLAG,
            INITIALIZE
            );


end structural_CPU_588;
```

Vhdl Simulation Report

Report Name: structural_CPU_588"
Kernel Library Name: <<RMILLER.STRUCTURAL>>STRUCTURAL_CPU_588
Kernel Creation Date: SEP-05-1990
Kernel Creation Time: 10:34:19
Run Identifer: 1
Run Date: SEP-05-1990
Run Time: 10:34:19

Report Control Language File: cpu.rcl
Report Output File : structural_cpu_588.rpt

Max Time: 9223372036854775807
Max Delta: 2147483646

Report Control Language :

```
Simulation_report structural_CPU_588 is
begin
report_name is "structural_CPU_588";
page_width is 120;
page_length is 40;
signal_format is horizontal;

sample_signals by_transaction in ns;
--sample_signals by_event in ns;
select_signal : Clock;
select_signal : instruction;
select_signal : zero_flag;
--select_signal /uut: Y0;
--select_signal /uut: Y1;
--select_signal /uut: Y2;
--select_signal /uut: Y3;
--select_signal /uut: Y4;
--select_signal /uut: Y5;
--select_signal /uut: Y6;
```

```
  --select_signal /uut: Y7;
  --select_signal /uut: Y8;
  --select_signal /uut: Y9;
  --select_signal /uut: Y10;
  --select_signal /uut: Y11;
  --select_signal /uut: Y12;
  --select_signal /uut: LOAD;
  --select_signal /uut: ADD;
  --select_signal /uut: BAND;
  --select_signal /uut: STORE;
  --select_signal /uut: JUMP;
  --select_signal /uut: JUMPZ;
  --select_signal /uut: COMP;
  --select_signal /uut: RSHIFT;
  --select_signal /uut: QWRITE;
  --select_signal /uut: QFETCH;
  --select_signal /uut: QREAD;
  select_signal : Control_good;
  select_signal : Control_bogus;

  end structural_CPU_588;
```


Report Format Information :

Time is in NS relative to the start of simulation
Time period for report is from 0 NS to End of Simulation
Signal values are reported by transaction (' ' indicates no transaction)

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
0	'U'	"UUU"	'U'	"XXXXXXXXXXXXX"	"XXXXXXXXXXXXX"
+1			'0'	"XXXXXXXXXXXXX"	"XXXXXX:XXXXXX"
1					
+1				"0010000000000"	"0010000000000"
2					
+1	'0'				
6					
+1	'1'			"ZZZZZZZZZZZZZ"	"ZZZZZZZZZZZZZ"
+2				"0000000001000"	"0000000001000"
10					
+1	'0'				
14					
+1	'1'				
+2				"0001000001000"	"0001000001000"
+3		"000"	'0'		
18					
+1	'0'				
22					
+1	'1'				
+2				"0100000000000"	"0100000000000"
26					
+1	'0'				
30					
+1	'1'				
+2				"0000010000000"	"0000010000000"
34					
+1	'0'				
38					
+1	'1'				
+2				"0000000001000"	"0000000001000"
42					
+1	'0'				

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
46					
+1	'1'				
+2				"0000000001000"	"0000000001000"
50					
+1	'0'				
54					
+1	'1'				
+2				"0000001000000"	"0000001000000"
58					
+1	'0'				
62					
+1	'1'				
+2				"0010000000000"	"0010000000000"
66					
+1	'0'				
70					
+1	'1'				
+2				"0000000001000"	"0000000001000"
74					
+1	'0'				
78					
+1	'1'				
+2				"0001000001000"	"0001000001000"
+3		"000"	'0'		
82					
+1	'0'				
86					
+1	'1'				
+2				"0100000000000"	"0100000000000"
90					
+1	'0'				
94					

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+1	'1'				
+2				"0000010000000"	"0000010000000"
98					
+1	'0'				
102					
+1	'1'				
+2				"0000000001000"	"0000000001000"
106					
+1	'0'				
110					
+1	'1'				
+2				"0000000001000"	"0000000001000"
114					
+1	'0'				
118					
+1	'1'				
+2				"0000001000000"	"0000001000000"
122					
+1	'0'				
126					
+1	'1'				
+2				"0010000000000"	"0010000000000"
130					
+1	'0'				
134					
+1	'1'				
+2				"0000000001000"	"0000000001000"
138					
+1	'0'				
142					
+1	'1'				
+2				"0001000001000"	"0001000001000"

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+3		"010"	'0'		
146					
+1	'0'				
150					
+1	'1'				
+2				"0100000000000"	"0100000000000"
154					
+1	'0'				
158					
+1	'1'				
+2				"0000010000000"	"0000010000000"
162					
+1	'0'				
166					
+1	'1'				
+2				"0000000001000"	"0000000001000"
170					
+1	'0'				
174					
+1	'1'				
+2				"0000000001000"	"0000000001000"
178					
+1	'0'				
182					
+1	'1'				
+2				"0000000000001"	"0000000000001"
186					
+1	'0'				
190					
+1	'1'				
+2				"0010000000000"	"0010000000000"
194					

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+1	'0'				
198					
+1	'1'				
+2				"0000000001000"	"0000000001000"
202					
+1	'0'				
206					
+1	'1'				
+2				"0001000001000"	"0001000001000"
+3		"011"	'0'		
210					
+1	'0'				
214					
+1	'1'				
+2				"0100000000000"	"0100000000000"
218					
+1	'0'				
222					
+1	'1'				
+2				"0000010000000"	"0000010000000"
226					
+1	'0'				
230					
+1	'1'				
+2				"0000000001000"	"0000000001000"
234					
+1	'0'				
238					
+1	'1'				
+2				"0000000001000"	"0000000001000"
242					
+1	'0'				

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
246					
+1	'1'				
+2				"0000000000010"	"0000000000010"
250					
+1	'0'				
254					
+1	'1'				
+2				"0010000000000"	"0010000000000"
258					
+1	'0'				
262					
+1	'1'				
+2				"0000000001000"	"0000000001000"
266					
+1	'0'				
270					
+1	'1'				
+2				"0001000001000"	"0001000001000"
+3		"001"	'0'		
274					
+1	'0'				
278					
+1	'1'				
+2				"0100000000000"	"0100000000000"
282					
+1	'0'				
286					
+1	'1'				
+2				"0000010000000"	"0000010000000"
290					
+1	'0'				
294					

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+1	'1'				
+2				"0000000100000"	"0000000100000"
298					
+1	'0'				
302					
+1	'1'				
+2				"0000000010000"	"0000000010000"
306					
+1	'0'				
310					
+1	'1'				
+2				"0000000010000"	"0000000010000"
314					
+1	'0'				
318					
+1	'1'				
+2				"0010000000000"	"0010000000000"
322					
+1	'0'				
326					
+1	'1'				
+2				"0000000001000"	"0000000001000"
330					
+1	'0'				
334					
+1	'1'				
+2				"0001000001000"	"0001000001000"
+3		"100"	'0'		
338					
+1	'0'				
342					
+1	'1'				

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+2				"0100000000000"	"0100000000000"
346					
+1	'0'				
350					
+1	'1'				
+2				"0000100000000"	"0000100000000"
354					
+1	'0'				
358					
+1	'1'				
+2				"0010000000000"	"0010000000000"
362					
+1	'0'				
366					
+1	'1'				
+2				"0000000001000"	"0000000001000"
370					
+1	'0'				
374					
+1	'1'				
+2				"0001000001000"	"0001000001000"
+3		"101"	'0'		
378					
+1	'0'				
382					
+1	'1'				
+2				"0100000000000"	"0100000000000"
386					
+1	'0'				
390					
+1	'1'				
+2				"0000100000000"	"0010100000000"

TIME (NS)	SIGNAL NAMES				
	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
394					
+1	'0'				
398					
+1	'1'				
+2				"0010000000000"	"0010000001000"
402					
+1	'0'				
406					
+1	'1'				
+2				"0000000001000"	"0001000001000"
410					
+1	'0'				
414					
+1	'1'				
+2				"0001000001000"	"0100000000000"
+3		"110"	'0'		
418					
+1	'0'				
422					
+1	'1'				
+2				"0100000000000"	"0000000000100"
426					
+1	'0'				
430					
+1	'1'				
+2				"0000000000100"	"0010000000000"
434					
+1	'0'				
438					
+1	'1'				
+2				"0010000000000"	"0000000001000"
442					

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+1	'0'				
446					
+1	'1'				
+2				"0000000001000"	"0001000001000"
450					
+1	'0'				
454					
+1	'1'				
+2				"0001000001000"	"0100000000000"
+3		"111"	'0'		
458					
+1	'0'				
462					
+1	'1'				
+2				"0100000000000"	"1000000000000"
466					
+1	'0'				
470					
+1	'1'				
+2				"1000000000000"	"0010000000000"
474					
+1	'0'				
478					
+1	'1'				
+2				"0010000000000"	"0000000001000"
482					
+1	'0'				
486					
+1	'1'				
+2				"0000000001000"	"0001000001000"
490					
+1	'0'				

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
494					
+1	'1'				
+2				"0001000001000"	"0100000000000"
+3		"000"	'0'		
498					
+1	'0'				
502					
+1	'1'				
+2				"0100000000000"	"0000010000000"
506					
+1	'0'				
510					
+1	'1'				
+2				"0000010000000"	"0000000001000"
514					
+1	'0'				
518					
+1	'1'				
+2				"0000000001000"	"0000000001000"
522					
+1	'0'				
526					
+1	'1'				
+2				"0000000001000"	"0000001000000"
530					
+1	'0'				
534					
+1	'1'				
+2				"0000001000000"	"0010000000000"
538					
+1	'0'				
542					

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+1	'1'				
+2				"0010000000000"	"0000000001000"
546					
+1	'0'				
550					
+1	'1'				
+2				"0000000001000"	"0001000001000"
554					
+1	'0'				
558					
+1	'1'				
+2				"0001000001000"	"0100000000000"
+3		"001"	'0'		
562					
+1	'0'				
566					
+1	'1'				
+2				"0100000000000"	"0 00010000000"
570					
+1	'0'				
574					
+1	'1'				
+2				"000( )10000000"	"0000000100000"
578					
+1	'0'				
582					
+1	'1'				
+2				"0000000100000"	"0000000010000"
586					
+1	'0'				
590					
+1	'1'				

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+2				"0000000010000"	"0000000010000"
594					
+1	'0'				
598					
+1	'1'				
+2				"0000000010000"	"0010000000000"
602					
+1	'0'				
606					
+1	'1'				
+2				"0010000000000"	"0000000001000"
610					
+1	'0'				
614					
+1	'1'				
+2				"0000000001000"	"0001000001000"
618					
+1	'0'				
622					
+1	'1'				
+2				"0001000001000"	"0100000000000"
+3		"010"	'0'		
626					
+1	'0'				
630					
+1	'1'				
+2				"0100000000000"	"0000010000000"
634					
+1	'0'				
638					
+1	'1'				
+2				"0000010000000"	"0000000001000"

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
642					
+1	'0'				
646					
+1	'1'				
+2				"0000000001000"	"0000000001000"
650					
+1	'0'				
654					
+1	'1'				
+2				"0000000001000"	"0000000000001"
658					
+1	'0'				
662					
+1	'1'				
+2				"0000000000001"	"0010000000000"
666					
+1	'0'				
670					
+1	'1'				
+2				"0010000000000"	"0000000001000"
674					
+1	'0'				
678					
+1	'1'				
+2				"0000000001000"	"0001000001000"
682					
+1	'0'				
686					
+1	'1'				
+2				"0001000001000"	"0100000000000"
+3		"011"	'0'		
690					

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+1	'0'				
694					
+1	'1'				
+2				"0100000000000"	"0000010000000"
698					
+1	'0'				
702					
+1	'1'				
+2				"0000010000000"	"0000000001000"
706					
+1	'0'				
710					
+1	'1'				
+2				"0000000001000"	"0000000001000"
714					
+1	'0'				
718					
+1	'1'				
+2				"0000000001000"	"0000000000010"
722					
+1	'0'				
726					
+1	'1'				
+2				"0000000000010"	"0010000000000"
730					
+1	'0'				
734					
+1	'1'				
+2				"0010000000000"	"0000000001000"
738					
+1	'0'				
742					

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+1	'1'				
+2				"0000000001000"	"0001000001000"
746					
+1	'0'				
750					
+1	'1'				
+2				"0001000001000"	"0100000000000"
+3		"100"	'1'		
754					
+1	'0'				
758					
+1	'1'				
+2				"0100000000000"	"0000100000000"
762					
+1	'0'				
766					
+1	'1'				
+2				"0000100000000"	"0010000000000"
770					
+1	'0'				
774					
+1	'1'				
+2				"0010000000000"	"0000000001000"
778					
+1	'0'				
782					
+1	'1'				
+2				"0000000001000"	"0001000001000"
786					
+1	'0'				
790					
+1	'1'				

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+2				"0001000001000"	"0100000000000"
+3		"101"	'0'		
794					
+1	'0'				
798					
+1	'1'				
+2				"0100000000000"	"0010100000000"
802					
+1	'0'				
806					
+1	'1'				
+2				"0000100000000"	"0010000001000"
810					
+1	'0'				
814					
+1	'1'				
+2				"0010000000000"	"0001000001000"
818					
+1	'0'				
822					
+1	'1'				
+2				"0000000001000"	"0100000000000"
826					
+1	'0'				
830					
+1	'1'				
+2				"0001000001000"	"0010100000000"
+3		"110"	'0'		
834					
+1	'0'				
838					
+1	'1'				

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+2				"0100000000000"	"0010000001000"
842					
+1	'0'				
846					
+1	'1'				
+2				"0000000000100"	"0001000001000"
850					
+1	'0'				
854					
+1	'1'				
+2				"0010000000000"	"0100000000000"
858					
+1	'0'				
862					
+1	'1'				
+2				"0000000001000"	"0000000000100"
866					
+1	'0'				
870					
+1	'1'				
+2				"0001000001000"	"0010000000000"
+3		"111"	'1'		
874					
+1	'0'				
878					
+1	'1'				
+2				"0100000000000"	"0000000001000"
882					
+1	'0'				
886					
+1	'1'				
+2				"1000000000000"	"0001000001000"

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
890					
+1	'0'				
894					
+1	'1'				
+2				"0010000000000"	"0100000000000"
898					
+1	'0'				
902					
+1	'1'				
+2				"0000000001000"	"1000000000000"
906					
+1	'0'				
910					
+1	'1'				
+2				"0001000001000"	"0010000000000"
+3		"000"	'1'		
914					
+1	'0'				
918					
+1	'1'				
+2				"0100000000000"	"0000000001000"
922					
+1	'0'				
926					
+1	'1'				
+2				"0000010000000"	"0001000001000"
930					
+1	'0'				
934					
+1	'1'				
+2				"0000000001000"	"0100000000000"
938					

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+1	'0'				
942					
+1	'1'				
+2				"0000000001000"	"0000010000000"
946					
+1	'0'				
950					
+1	'1'				
+2				"0000001000000"	"0000000001000"
954					
+1	'0'				
958					
+1	'1'				
+2				"0010000000000"	"0000000001000"
962					
+1	'0'				
966					
+1	'1'				
+2				"0000000001000"	"0000001000000"
970					
+1	'0'				
974					
+1	'1'				
+2				"0001000001000"	"0010000000000"
+3		"001"	'1'		
978					
+1	'0'				
982					
+1	'1'				
+2				"0100000000000"	"0000000001000"
986					
+1	'0'				

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
990					
+1	'1'				
+2				"0000010000000"	"0001000001000"
994					
+1	'0'				
998					
+1	'1'				
+2				"0000000100000"	"0100000000000"
1002					
+1	'0'				
1006					
+1	'1'				
+2				"0000000010000"	"0000010000000"
1010					
+1	'0'				
1014					
+1	'1'				
+2				"0000000010000"	"0000000100000"
1018					
+1	'0'				
1022					
+1	'1'				
+2				"0010000000000"	"0000000010000"
1026					
+1	'0'				
1030					
+1	'1'				
+2				"0000000001000"	"0000000010000"
1034					
+1	'0'				
1038					
+1	'1'				

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+2				"0001000001000"	"0010000000000"
+3		"010"	'1'		
1042					
+1	'0'				
1046					
+1	'1'				
+2				"0100000000000"	"0000000001000"
1050					
+1	'0'				
1054					
+1	'1'				
+2				"0000010000000"	"0001000001000"
1058					
+1	'0'				
1062					
+1	'1'				
+2				"0000000001000"	"0100000000000"
1066					
+1	'0'				
1070					
+1	'1'				
+2				"0000000001000"	"0000010000000"
1074					
+1	'0'				
1078					
+1	'1'				
+2				"0000000000001"	"0000000001000"
1082					
+1	'0'				
1086					
+1	'1'				
+2				"0010000000000"	"0000000001000"

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
1090					
+1	'0'				
1094					
+1	'1'				
+2				"0000000001000"	"0000000000001"
1098					
+1	'0'				
1102					
+1	'1'				
+2				"0001000001000"	"0010000000000"
+3		"011"	'1'		
1106					
+1	'0'				
1110					
+1	'1'				
+2				"0100000000000"	"0000000001000"
1114					
+1	'0'				
1118					
+1	'1'				
+2				"0000010000000"	"0001000001000"
1122					
+1	'0'				
1126					
+1	'1'				
+2				"0000000001000"	"0100000000000"
1130					
+1	'0'				
1134					
+1	'1'				
+2				"0000000001000"	"0000010000000"
1138					

TIME	SIGNAL NAMES				
(NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
+1	'0'				
1142					
+1	'1'				
+2				"0000000000010"	"0000000001000"
1146					
+1	'0'				
1150					
+1	'1'				
+2				"0010000000000"	"0000000001000"
1154					
+1	'0'				
1158					
+1	'1'				
+2				"0000000001000"	"0000000000010"
1162					
+1	'0'				
1166					
+1	'1'				
+2				"0001000001000"	"0010000000000"
+3		"100"	'1'		
1170					
+1	'0'				
1174					
+1	'1'				
+2				"0100000000000"	"0000000001000"
1178					
+1	'0'				
1182					
+1	'1'				
+2				"0000100000000"	"0001000001000"
1186					
+1	'0'				

structural_CPU_588"

TIME (NS)	CLOCK	INSTRUCTION(2 DOWNTO 0)	ZERO_FLAG	CONTROL_GOOD(12 DOWNTO 0)	CONTROL_BOGUS(12 DOWNTO
1190					
+1	'1'				
+2				"0010000000000"	"0100000000000"
1194					
+1	'0'				
1198					
+1	'1'				
+2				"0000000001000"	"0000100000000"
1202					
+1	'0'				
1206					
+1	'1'				
+2				"0001000001000"	"0010000000000"
+3		"101"	'1'		
1210					
+1	'0'				
1214					
+1	'1'				
+2				"0100000000000"	"0000000001000"
1218					
+1	'0'				
1222					
+1	'1'				
+2				"0010000000000"	"0001000001000"
1226					
+1	'0'				
1230					
+1	'1'				
+2				"0000000001000"	"0100000000000"
1234					
+1	'0'				
1238					

The following two files reflect the Unix script file and its results which automatically processed the two cpu VHDL files invoking pre_verif and verif.

```
date

time steed -vhdl -enum cpu_588s.vhd

time mv verif.input cpu_588s.verif

time steed -vhdl -enum cpu_588s_bogus.vhd

time mv verif.input cpu_588s_bogus.verif

time /olympus3/eng/rlmiller/verif/verif cpu_588s.verif cpu_588s_bogus.verif

date
```

```
olympus> !cpu
cpu_588_verif_timer  > timerstuff

Tues Oct 30 10:34:55 EDT 1990

# steed -vhdl -enum cpu_588s.vhd
#Circuit Summary:
# Entity name    : CPU_CONTROLLER1
# Architecture   : STRUCTURAL
#---------------
#number of gates = 42
#number of wires = 62
#number of inputs = 4
#number of outputs = 13
#number of latches = 16

#steed: cputime for reading in circuit: 0.3s 0.3s
#steed: cputime for levelling circuit: 0.0s 0.4s
#steed: cputime for rearranging gate inputs: 0.0s 0.4s
#steed: cputime for creating dummy gates: 0.0s 0.4s
#number of equivalent faults = 190
#steed: cputime for generating fault list: 0.0s 0.4s
#steed: cputime for miscellaneous allocation: 0.0s 0.4s
#Memory required for all covers = 438.000000 bytes
#steed: cputime for generating the partial covers : 0.5s 0.9s
#steed: cputime for all processes: 0.0s 0.9s
OUTPUT PRODUCED IN FILE verif.input
        1.6 real         0.9 user         0.4 sys
        0.1 real         0.0 user         0.0 sys


# steed -vhdl -enum cpu_588s_bogus.vhd
#Circuit Summary:
# Entity name    : CPU_CONTROLLER1
# Architecture   : STRUCTURAL_bogus
```

```
#---------------
#number of gates = 42
#number of wires = 62
#number of inputs = 4
#number of outputs = 13
#number of latches = 16

#steed: cputime for reading in circuit: 0.4s 0.4s
#steed: cputime for levelling circuit: 0.0s 0.4s
#steed: cputime for rearranging gate inputs: 0.0s 0.4s
#steed: cputime for creating dummy gates: 0.0s 0.4s
#number of equivalent faults = 190
#steed: cputime for generating fault list: 0.0s 0.4s
#steed: cputime for miscellaneous allocation: 0.0s 0.4s
#Memory required for all covers = 438.000000 bytes
#steed: cputime for generating the partial covers : 0.5s 0.9s
#steed: cputime for all processes: 0.0s 0.9s
OUTPUT PRODUCED IN FILE verif.input

        1.6 real         0.9 user         0.5 sys
        0.1 real         0.0 user         0.0 sys


# Machine 1 inputs  4 outputs 13 latches 16
# Machine 2 inputs  4 outputs 13 latches 16
#Time to read in covers : 8.800000e-01 secs
#MACHINES ARE DIFFERENT
#THE DISTINGUISHING SEQUENCE IS :
--1-
--1-
--1-
--1-
1010
----
Number of states = 15
Number of edges  = 36
Number of entries  = 9
Number of save_difs  = 0
#Time for verification  : 3.300000e-01 secs
#Total user time        : 1.210000e+00 secs

Tues Oct 30 10:35:04 EDT 1990

olympus>
```

The following two files reflect the unix script file and its result which automatically invoked the VHDL software support environment in order to process the cpu VHDL files through the VHDL simulator.

```
date

time vhdl cpu_588s.vhd

time mg "cpu_controller1(structural)"

time vhdl cpu_588s_bogus.vhd

time mg "cpu_controller1(structural_bogus)"

time vhdl testbench_cpu

time mg -top "test_bench(structural_cpu_588)"

time build -replace "test_bench(structural_cpu_588)"

time sim structural_cpu_588

time rg structural_cpu_588 cpu.rcl

date
```

```
atlas% cpu_588_timer
Wed Sep  5 10:09:53 EDT 1990
Standard VHDL 1076 Support Environment Version 2.1 - 1 February 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

       73.6 real        46.2 user         6.2 sys
Standard VHDL 1076 Support Environment Version 2.1 - 1 February 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

      554.7 real       452.7 user        30.8 sys
Standard VHDL 1076 Support Environment Version 2.1 - 1 February 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

       58.4 real        38.7 user         5.0 sys
Standard VHDL 1076 Support Environment Version 2.1 - 1 February 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

      505.8 real       445.5 user        24.1 sys
Standard VHDL 1076 Support Environment Version 2.1 - 1 February 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

       57.3 real        30.6 user         6.1 sys
Standard VHDL 1076 Support Environment Version 2.1 - 1 February 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

      135.1 real        97.9 user         8.7 sys
Standard VHDL 1076 Support Environment Version 2.1 - 1 February 1990
```

```
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

       62.4 real        28.6 user         9.3 sys
Standard VHDL 1076 Support Environment Version 2.1 - 1 February 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

%VHDSIM-N-SIGTRAN Signal Tracing turned on
%VHDSIM-N-SIGTRAN Signal Tracing turned on after 0 fs

%VHDSIM-N-TRANSOV Transaction Limit Exceeded after 350 ns

%VHDSIM-N-TERMINA Explicit Termination requested after 1238 ns

      205.0 real       180.3 user         5.5 sys
Standard VHDL 1076 Support Environment Version 2.1 - 1 February 1990
Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.

       55.9 real        39.2 user        11.3 sys
Wed Sep  5 10:38:32 EDT 1990
atlas%
```

Bibliography

1. Advanced Tactical Fighter Special Program Office, Aeronautical Systems Division, Air Force Systems Command. Contract F33657/2040-2-A with ZYCAD Corporation. Wright-Patterson AFB OH, February 1990.

2. Department of Defense, "Joint Integrated Avionics Plan for New Aircraft," Industry Copy, prepared under direction of Office of the Under Secretary of Defense Acquisition, March 1989.

3. Z. Kohavi, *Switching and Finite Automata Theory, Second Edition.* New York, McGraw Hill Book Company, 1978.

4. T. Booth, *Sequential Machines and Automata Theory.* New York, Wiley and Sons, Inc, 1967.

5. D. Barton, "Examples of Formally Verified Circuits," in *Proc, 1990 VHDL Users' Group Fall Meeting*, IEEE Press, October, 1990.

6. A. Ghosh and others, "Verification of Interacting Sequential Circuits," Unpublished Manuscript, University of California, Berkeley, 1989.

7. S. Devadas, H-K. T. Ma, and A. R. Newton. "On the Verification of Sequential Machines at Differing Levels of Abstraction," *IEEE Transactions on CAD*, Vol 7, No. 6, IEEE Press, June 1988.

8. T. J. Wagner, "Verification of Hardware Designs through Symbolic Manipulation," in *Proc. Design Automation and Microprocessors Symposium*, (Palo-Alto, California), IEEE Press, February 1977.

9. K. J. Supowit and S. J. Friedman, "A new method for verifying sequential circuits," in *Proc, 23rd Design Automation Conf.*, IEEE Press, June 1986.

10. J. R. Burch and others, "Sequential Circuit Verification Using Symbolic Model Checking," In *Proceedings of Design Automation Conf.*, IEEE Press, 1990.

11. B. Moszkowski, "A Temporal Logic for Multilevel Reasoning about Hardware." *IEEE Computer*, IEEE Press, February 1985.

12. M. C. Browne and others, "Automatic Verification of Sequential Circuits Using Temporal Logic," *IEEE Trans. on Computers*, Vol C-35, No. 12, IEEE Press, December 1986.

13. S. Kimura and E. M. Clarke, "A Parallel Algorithm for Constructing Binary Decision Diagrams," Unpublished Manuscript, Carnegie-Mellon University, undated.

14. M. C. Browne and others, "CSML Language Reference, " Unpublished Manuscript, Carnegie-Mellon University, May 1989.

15. J. R. Armstrong, *Chip-Level Modeling with VHDL.* Englewood Cliffs, New Jersey, Prentice Hall, 1989.

16. IEEE, Computer Society Standards Committee, *IEEE Standard VHDL Language Reference Manual,* ANSI/IEEE Std 1076-1987 IEEE Press, New York, New York, 1988.

17. E. M. Clarke and others, "A Language for Compositional Specification and Verification of Finite State Hardware Controllers, " CMU-CS-89-110, Dept. of Computer Science, Carnegie-Mellon University, January 1989.

18. D. Barton, "Examples of Formally Verified Circuits," in *Proc, 1989 VHDL Users' Group Fall Meeting*, IEEE Press, October, 1989.

19. *ExpressVHDL.* Company Brochure, i-Logix Inc., Burlington, Massachusetts, 1990.

20. R. Lipsett and others, *VHDL: Hardware Description and Design.* Massachussetts: Kluwer Academic Publishers, 1989.

21. _____, ZYCAD VHDL Users's Manual, Revision 2.0a, Federal Services Group, ZYCAD Corporation, Mount Olive, New Jersey,1990.

22. J. P. Hayes, *Computer Architecture and Organization* (Second Edition). New York: McGraw Hill Book Company, 1988.

23. _____, Joint Integrated Avionics Working Group TM_bus (JTM-bus) Specification DOCUMENT J89-N1B, CAB III REV 1/30, April 1990.

Vita

Captain Richard L. Miller [redacted]o. He graduated from high school in Burton, Ohio, June 1974, and entered the Air Force as an Avionics Navigation Systems Specialist in December 1977. In December, 1984 he received a Bachelor of Electrical Engineering degree from Auburn University, Auburn, Alabama through the Air Force's Airman Education and Comissioning Program. He received his comission upon completion of Officer Training School in April, 1985. From May, 1985 until June, 1989, he served first as a microelectronics design engineer and then as executive officer for the Avionics Laboratory, Wright Research and Development Center, Aeronautical System's Division, Wright-Patterson AFB. He entered the School of Engineering, Air Force Institute of Technology in June, 1989.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Master's Thesis

4. TITLE AND SUBTITLE

SPECIFICATION AND EQUIVALENCE VALIDATION OF SEQUENTIAL CIRCUITS VIA VHDL.

5. FUNDING NUMBERS

6. AUTHOR(S)

Richard L. Miller, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GE/ENG/90D-41

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

ASD/YF
WPAFB OH 45433-6583

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited
3 Dec 90.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This research presents a merger of the specification and design capabilities of the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) with a known verification method (UC Berkeley's verif software) in order to solve the design and verification problem of sequential circuits. The fruits of this research are a behavioral VHDL model for sequential circuit specification, a structural VHDL model for sequential circuit design, and a method for comparing two circuits described using these VHDL models in order to demonstrate circuit equivalence. The behavioral and structural VHDL models were developed and tested within the Intermetric's VHDL software support environment. Modifications were made to the existing UC Berkeley verif software so that it could accept sequential circuits described using the structural VHDL model. Additionally, a behavioral to structural VHDL translator (b2s) was developed such that sequential circuits expressed in the behavioral VHDL model could be shown equivalent to structural VHDL designs via the UC Berkeley verification software.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Sequential Circuits, VHDL, Verification, Circuit Equivalence Specification, Modeling	310
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to **stay within the lines to meet optical scanning requirements.**

Block 1. Agency Use Only (Leave Blank)

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Names(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency. Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of ..., To be published in .... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availablity Statement. Denote public availability or limitation. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR)

DOD	- See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	- See authorities
NASA	- See Handbook NHB 2200.2.
NTIS	- Leave blank.

Block 12b. Distribution Code.

DOD	- DOD - Leave blank
DOE	- DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports
NASA	- NASA - Leave blank
NTIS	- NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.